TEXAS INSTRUMENTS

The proLogic™ Compiler

User's Guide

1991

The proLogic™ Compiler User's Guide

TEXAS INSTRUMENTS

IVAN 997 5604 PERSONAL

FAX 997 5650

IMPORTANT NOTICE

proLogic compiler, Release 2.00

This logic compiler was prepared by proLogic Systems Inc. for distribution by Texas Instruments Incorporated, expressly to support devices manufactured by them.

Assistance on programming Texas Instruments Programmable Logic is also available, upon request, from the nearest TI field sales office, local authorized TI distributor, or by calling the Texas Instruments PLD Hotline (214) 997-5666, or Texas Instruments PLD Bulletin Board Service (214) 997-5665. [BBS Information: 1200/2400, 8, N, 1]

TRADEMARKS

5

Read This First

This Preface contains a brief description of the contents of *The proLogic Compiler User's Guide.*

How to Use this Manual

Preface

The proLogic compiler is a software development design tool used to program Texas Instrument Programmable Logic Devices (PLD). This development software package quickly converts your logic design to a JEDEC fuse map that can be downloaded to a device programmer. proLogic allows you to describe your logic design in any of the following formats:

- Truth Table
- Boolean Equations
- State Diagrams

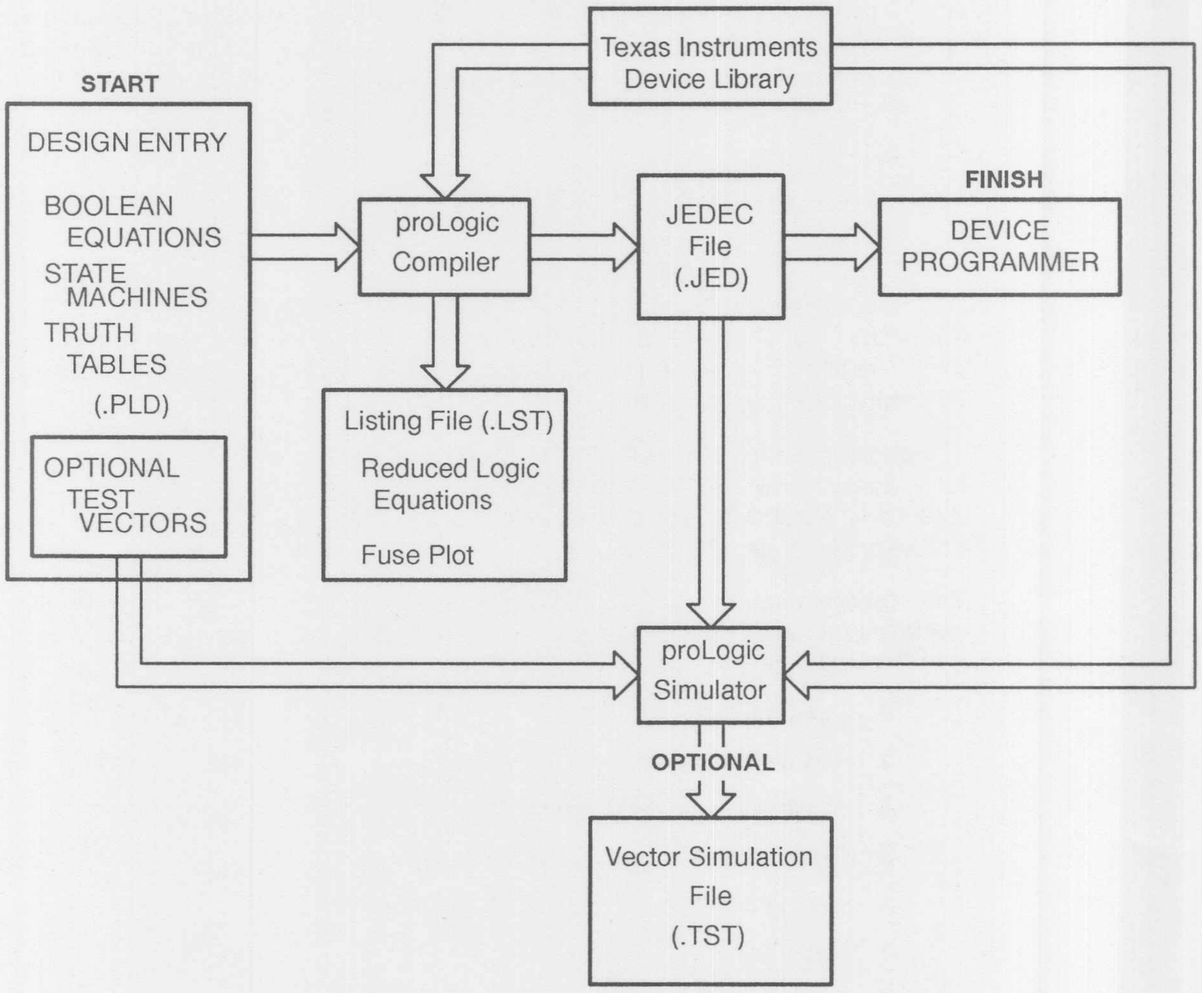
From your logic design, the proLogic compiler will create a standard JEDEC fuse map. The JEDEC file can be downloaded to a device programmer to produce a functional programmed device. Many Texas Instruments authorized distributors provide these programming services.

The proLogic compiler also serves as a functional test vector simulator. The simulator uses the fuse list portion from the JEDEC file to create a functional device model. It can then execute the simulation vectors against this model. The results are automatically placed in a file for evaluation.

The proLogic compiler, when combined with a device programmer, will allow you to create integrated circuits to your own specifications. The advantages to this new design methodology are numerous:

- Lower chip count
- Reduced board space
- Lower power requirements
- Higher reliability (fewer interconnects)
- Proprietary design protection (fuse protection)
- Fewer parts in inventory
- Greater design flexibility

The proLogic compiler is complete and ready for installation on your IBM™ compatible PC. The following block diagram shows the steps from design to proLogic to a programmed device.



proLogic Block Diagram

Section 1

How To Begin

The proLogic Compiler Package contains:

- ❏ a proLogic Compiler User's Guide
- ❏ one proLogic Diskette

The proLogic Diskette contains:

- ❏ the executable files
- ❏ Header files (.H) which apply to multiple Programmable Logic Devices
- ❏ the Texas Instruments PLD Architecture files

Installation

Step 1: On your IBM or IBM compatible PC, create a new directory on the hard disk and name it PROLOGIC. < `C:\MD PROLOGIC` >

Step 2: Copy all of the files on the proLogic diskettes into the proLogic directory.

Step 3: Enter < `PROLOGIC` > at the `C:\PROLOGIC` prompt to self-extract the files.

Step 4: The file, NAND3.PLD is a sample Application PLD (sample program source file) needed to describe the PLDs.

Compile the sample device specification by entering the command.

```
LC NAND3
```

Compiling NAND3 produces the output files NAND3.JED and NAND3.LST. The JEDEC file (NAND.JED) is used to program the device. The listing file (NAND.LST) is a fuse plot showing the programmed device.

Step 5: Simulate device programming and testing by entering the command

```
LS NAND3
```

Logic Simulation uses NAND3.JED to produce the output file NAND3.TST. All of these files including the source file, are text files so you can print, type, or edit them to see the results.

Section 2

Programmable Logic

A Programmable Logic Device (PLD) is a type of integrated circuit whose function is field-configured. These devices allow you to create integrated circuits to your own specifications at a reasonable cost, thus reducing the chip count.

Programmable Logic Devices have the same basic digital building blocks that Small Scale Integration (SSI) devices have. The difference between SSI devices and PLDs is primarily the higher levels of integration in PLDs.

There are more input signals per gate. Even a common PLD like the TIBPAL16R4 has 32-input AND gates.

There are more gates per device. The 16R4 has sixty-four 32-input AND gates. Due to the great number of AND gates and associated inputs, a new notation is required to simplify the logic diagrams.

Conventional schematics tend to flow from left to right, with input signals on the left and output signals on the right. Thus a three input AND gate is drawn.

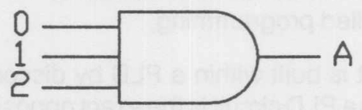


Figure 1. SSI Three Input AND Gate

The new PLD logic notation retains the convention of input signals on the left and output signals on the right. The change is that the input signals flow into a gate vertically from the top or bottom. A PLD AND gate is drawn

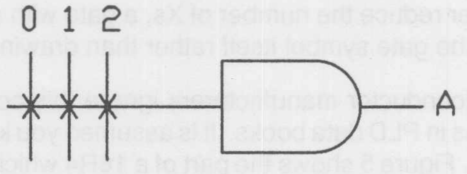


Figure 2. PLD Three Input AND Gate

These PLD AND gates are also called product terms. A product term refers to any n-input AND gate. The main reason for vertical inputs is that PLDs have a very regular structure. In the 16R4, all 32 of the input signals are input to each of the 64 AND gates. This permits the whole logic structure of the device to be concisely noted in tabular format as shown in Figure 3. The "x" at each intersection indicates that a programmable fuse is connecting each input line to the input of the AND gate.

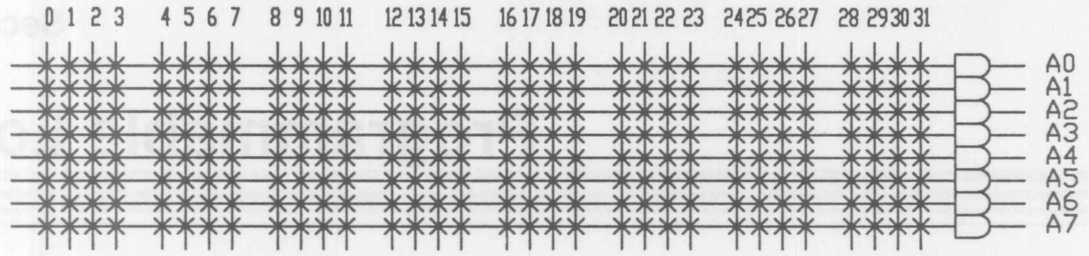


Figure 3. Eight 32-Input AND Gates

The 16R4 PLD is a 20 pin device. Figure 4 shows how the sixty-four 32-input AND gates are interfaced to the I/O pins. Note that the "x"s have been removed. The fuse still exist, however the "x"s are not shown in order to simplify the diagram.

Figure 4 shows that the device in its unprogrammed state. The output of all 64 AND gates is logic LOW because the input buffers feed both the true and the complement of all inputs into all the AND gates. In Boolean Logic notation:

```
a & !a = 0
```

for all AND gate output.

The basic difference between building a circuit using SSI devices and building a circuit with PLDs is the way in which the gates form a circuit. With SSI devices the gates are connected with wires or traces. Within a PLD, the gates are connected by a process called programming.

A circuit is built within a PLD by disconnecting inputs from gates. In this respect, building a PLD circuit is the exact opposite of building an SSI circuit. A PLD in its initial state has all possible gate connections already made. Programming consists of removing the connections that are not needed for your design. The connections that remain define function of the programmed device .

The notation used to show a programmed device is to draw Xs where connections remain, and draw nothing where connections have been removed. (In most programmed PLDs the number of connections remaining after programming is a lot less than the ones removed, so this convention results in a less cluttered diagram.) To further reduce the number of Xs, a gate with all connections intact is drawn with an X in the gate symbol itself rather than drawing individual Xs at each gate input.

Semiconductor manufacturers ignore this convention when printing the logic diagrams in PLD data books. It is assumed you know that an X is implied at all intersections. Figure 5 shows the part of a 16R4 which has been programmed to implement a 3-input NAND gate. The output is on pin19. The inputs are on pin2, pin3, and pin4.

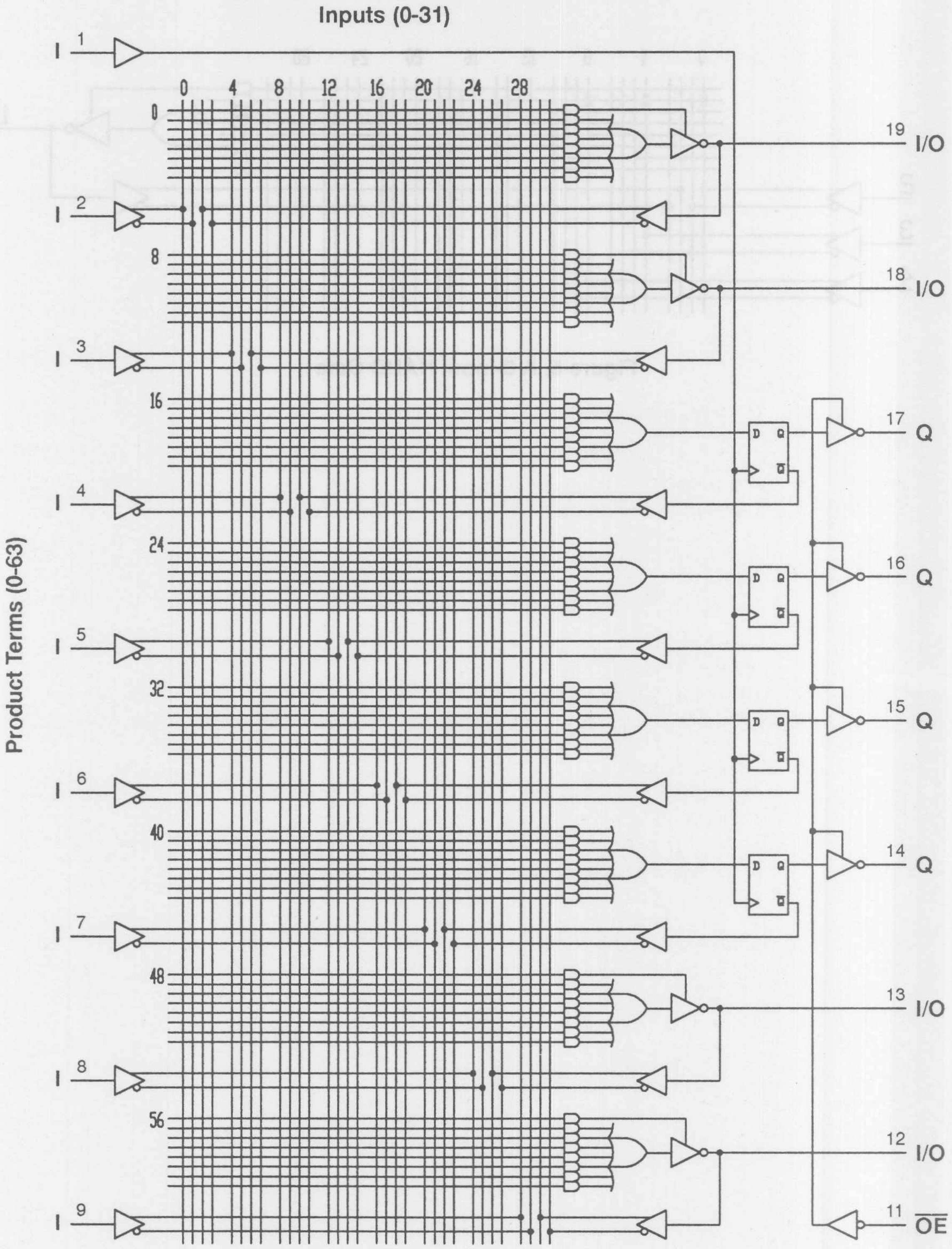


Figure 4. 16R4 Logic Diagram

Figure 5 also illustrates one other PLD feature which you should know. The PLD gates are implemented so that when all gate inputs are disconnected, the gate output is asserted. Figure 5 shows the output buffer for pin 19 as always enabled.

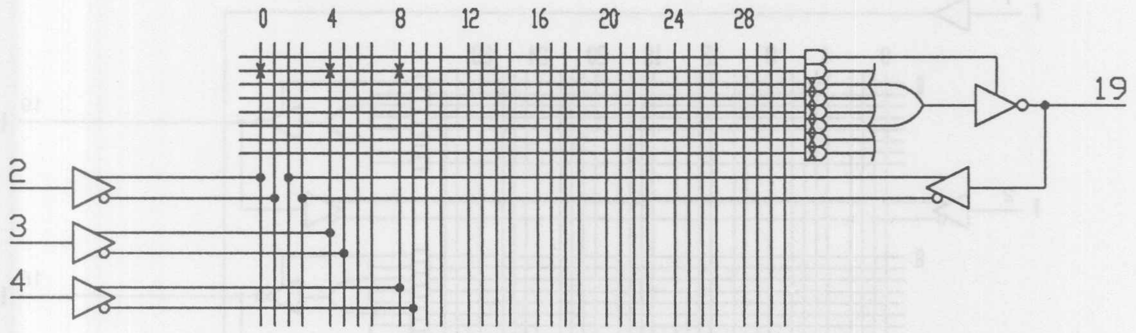


Figure 5. A 3-input NAND Gate

Section 3

PLD Design Implementation

Device Programmers

A device programmer is an electronic machine which programs the specified cells of a programmable device. Since PLDs are available in many technologies, a programming algorithm is defined for each type of device. The number of different programming algorithms is relatively large because algorithms can vary by both product technology and manufacturer. These algorithms involve voltages and currents not used during normal device operation.

Texas Instruments continually evaluates new programming equipment. TI Programmable Logic data books are a good reference source for approved device programmer vendors.

JEDEC Files

All device programmers accept an input file in a standard format. This file is called a JEDEC file because its format was developed by the Joint Electronic Device Engineering Council (JEDEC). JEDEC files, being comprised of ASCII characters, are also readable as text files. This example of a fuse list line of a JEDEC-standard file

```
L040 10011001010101101111*
```

is interpreted by the device programmer to mean:

1. program cell 40. The string of 1s and 0s which follow the L040 define the value of consecutive cells after programming. 1 means program the cell. The L040 gives the decimal address of the first cell in the string defined by the line.

2. do not program cell 41. The second digit in the string is a 0 which means do not program the cell.

3. program cells 43, 44, 47, 49, 51, 53, 54 and 56 through 59.

Another kind of JEDEC-standard line is this example of a test vector:

```
V01 NNN010110N10LLLHHNNN*
```

These lines are used by the device programmer to functionally test the programmed device. The V01 is a sequence number. The other characters are called test conditions. The first applies to device pin 1, the second to pin2, and so forth. This sample shows a 20-pin device. Test condition N means do nothing, so the first 0 means apply logic low voltage to pin 4. 1 applies logic high and L and H test for output level low and high.

The proLogic Compiler

The proLogic Compiler is an interface to a device programmer. It accepts your specifications in symbolic form, manages the task of specifying each of the thousands of cell states, and produces a JEDEC file to instruct the device programmer.

To see how it works, let's take the example at the end of Section 2. It shows a '16R4 programmed to implement a 3-input NAND gate. The output is to be pin19. The inputs are on pin2, pin3 and pin4. A PLD specification file must be generated before a JEDEC file can be produced to program the device.
The PLD specification is a simple text file which tells proLogic which cells to program in the PLD. The proLogic compiler takes its input from a source file in the same way that a compiler for microprocessors does. In order to program a '16R4 so that pin19 is a 3-input NAND gate, we need a source file called NAND3.PLD which has the following three lines:

```
include p16r4;
!pin19 = pin2 & pin3 & pin4;
pin19.oe = 1;
```

The first line specifies the PLD to be programmed. The other lines specify the two signals required to implement the circuit.

Once NAND3.PLD is available, entering the DOS command

```
LC NAND3
```

executes the proLogic Compiler. The execution result is a JEDEC file named NAND3.JED which specifies the 61 unwanted connections to be programmed by the device programmer. The following is the part of the file which has the fuse list lines.

```
proLogic Compiler (JEDEC Object A100) V2.00
Serial bxav0
Copyright (C) 1988 proLogic Systems Inc.

    p16r4 revision 89.2.11

*N_csidp16r4
*QP20
*QF2048
*F0
*L0000 11111111111111111111111111111111
*L0032 01110111011111111111111111111111
*C07E6
```

While generating the JEDEC file, the compiler may find errors in your program. When it does, it describes what seems to be wrong on the computer display. The error text completely describes the problem, so there is no need for you to look up error codes in the manual. Another file called NAND.LST is also created which allows you to read the NAND3.JED file. The part of the file which describes pin19 of the 16R4 looks like this:

```
  proLogic Compiler (Fuse Plot A100) V2.00
  Serial bxaw0
  Copyright (C) 1991 proLogic Systems Inc.

  p16r4 revision 89.2.11

                  11 1111 1111 2222 2222 2233
      0123 4567 8901 2345 6789 0123 4567 8901

    0 ---- ---- ---- ---- ---- ---- ---- ----    OE
    1 X--- X--- X--- ---- ---- ---- ---- ----    +
    2 XXXX XXXX XXXX XXXX XXXX XXXX XXXX XXXX    +
    3 XXXX XXXX XXXX XXXX XXXX XXXX XXXX XXXX    +  !pin19
    4 XXXX XXXX XXXX XXXX XXXX XXXX XXXX XXXX    +
    5 XXXX XXXX XXXX XXXX XXXX XXXX XXXX XXXX    +
    6 XXXX XXXX XXXX XXXX XXXX XXXX XXXX XXXX    +
    7 XXXX XXXX XXXX XXXX XXXX XXXX XXXX XXXX    +

      | |  | |  | |  | |  | |  | |  | |  | |
   pin2 |  3 |  4 |  5 |  6 |  7 |  8 |  9 |
        |    |    |    |    |    |    |    |
      pin1   18   17   16   15   14   13   11

   Legend:
X  : Cell intact       (JEDEC 0)
-  : Cell programmed   (JEDEC 1)

X- : True input term
-X : Complement input term
XX : Any XX pair in a product term yields product term LOW.
-- : No input term (don't care).  A product term comprised
     entirely of -- yields product term HIGH.
```

Compare the NAND3.LST file to Figure 5. This format, called a fuse plot, is just a printer oriented version of the logic diagram. It has some new words in it – an input term is another word for a signal; a product term is another word for AND gate. Other than that, the fuse plot is about the same as a logic diagram.

The Compiler Command Line

When you run the compiler, you need to specify the name of the program source file as the first DOS command line argument. There must be at least one space between the LC command and the source file-name. When no file extension is specified (.PLD) is assumed. Following the source file name, you can have zero or more command-line options. Options must also be separated from each other and from the file-name by at least one space.

Command Line Options

The -**a** Option

Causes all attributes and functions to be included in the function set listing.

The -**f** Option

No fuse plot is created as part of the listing (.LST) file.

The -**I** Option

–Ipath–name[;path–name]...

proLogic is able to search multiple directories for include files. Each path-name specifies a directory to be searched in addition to the current directory. They are searched in the indicated order when multiple directories are specified.

The -**v** Option

Causes the compiler to report its current status on the computer display as it compiles a program.

Example:

The command line

```
lc nand3 -f- -v -I\header;\p16
```

compiles the source file NAND3.PLD, creating a listing file NAND3.LST which has no fuse plot map. While compiling, it reports status on your PC monitor. If an include file cannot be found in the current directory, the directory whose path is \HEADER is first searched for the file and if not found there, the directory whose path is \P16 is searched.

The proLogic Simulator

The proLogic Simulator is a software version of a device programmer. To load it, enter the DOS command

```
LS NAND3
```

The simulator uses the fuse list lines from the JEDEC file (NAND3.JED) to create a functional device model. It then executes the test vectors against this model. The test results are placed in file NAND3.TST:

```
proLogic Simulator V2.00
Copyright (C) 1991 proLogic Systems Inc.

Architecture Description: p16r4.lxa
JEDEC Fuse Information:   nand3.jed
JEDEC Test Vectors:       nand3.jed

V1     N000 NNNN NNNN NNNN NNHN
V2     N001 NNNN NNNN NNNN NNHN
V3     N010 NNNN NNNN NNNN NNHN
V4     N011 NNNN NNNN NNNN NNHN
V5     N100 NNNN NNNN NNNN NNHN
V6     N101 NNNN NNNN NNNN NNHN
V7     N110 NNNN NNNN NNNN NNHN
V8     N111 NNNN NNNN NNNN NNLN

No errors detected with 8 Test Vectors.
```

You will find the same test vectors in NAND3.JED. The more readable source which created them is in NAND3.PLD.

In the example, no errors were found by the simulator. When errors exist, the simulator notes them with a single character code. At the end of the test listing, it appends a legend to explain the meaning of the error codes. This puts all the information you need in one place. There is no need to refer to the manual for a description of the error codes.

proLogic Simulator Input Files

When you run the simulator, you will need to supply the name of the Test Vectors file. It is a text file in JEDEC 3-A format which contains the test vectors (JEDEC field V) to be executed against the compiled device model. The DOS path name of the test vector file is the first parameter of the DOS command as shown in these examples:

```
LS VECTORS
LS TEST.JED
LS \TEST\VECTORS.VF1
```

When no file extension is specified, proLogic defaults to (.JED).

If your Test Vectors were created by the proLogic Compiler, that is the only parameter required. If not, the following are other options which can be used.

The - a Option

This option specifies the DOS path name of the Architecture Description file. This file contains a description of the programmable device to be simulated. The proLogic simulator uses this file and the Fuse Information file to compile an optimized device model prior to beginning actual simulation. Example:

```
LS NAND3 -A\LXA\P16R4
```

Architecture Description files always have a (.LXA) file extension. When the - a option is not specified, proLogic examines the Fuse Information file for a field beginning with the characters N_csid. If found, the remainder of the field characters identify the Architecture Description file.

The - j Option

This option specifies the DOS path name of the Fuse Information file. It is also a text file in JEDEC 3-A format. It contains the programming information (JEDEC field F) which specifies the device function. Example:

```
LS VECTORS -JFUSES.JED
```

When no file extension is specified, proLogic defaults to (.JED). When the - j option is not supplied, proLogic assumes the fuse information is contained in the Test Vectors file.

The –n Option

This option inverts the normal power up condition of the registers. This can be used to simulate with true power up conditions the TIBPAL16RX–15 and –25 devices, along with the TIB82S105 and TIB82S167 devices. Example:

```
LS 74374 -N
```

When this option is not specified, the registers will power up with a logic '0' which is how the IMPACT-X™ technology PLDs and EPLDs power up.

The proLogic Simulation Algorithm

The simulator behaves just like an electronic device tester. The next few paragraphs describes the device tester as a piece of hardware. These are the details you need to know to test your program.

Before executing any test vectors, the simulator removes voltage from all device pins. It next applies test condition 0 to the device ground pin and 1 to the device V_{CC} pin to simulate power-on. All memory elements power-on clear unless the device manufacturer specifies that they power-on set or the –n option is used. Note that power-on or -off may be simulated by a test vector which applies test conditions to the power and ground pins.

After power-on, the simulator executes each test vector in the order encountered in the Test Vectors file. Execution consists of performing these four steps in order:

1. Remove voltage from test pins (L,H,Z) in pin number sequence.
2. Apply input and I/O pin voltage (0..9,F,X) in pin number sequence excepting clock (C,K).
3. Apply clock pin voltage (C,K) in pin number sequence.
4. Read test pin voltage (L,H,Z).

Input pins with no voltage applied are assumed to float logic high. Applied voltages persist from test vector to test vector when neither driven nor tested (N).

Section 4

proLogic Diagrams

A 3-input NAND Gate

The source file NAND3.PLD which programs 16R4 pin 19 to be a 3-input NAND gate has these three lines:

```
include p16r4;
!pin19 = pin2 & pin3 & pin4;
pin19.oe = 1;
```

The first line names the the header file (.H) for the PLD. The other two lines are called assignment statements.

An assignment has an expression made up of signal names and gate operators. Signal names identify signals internal to the PLD to be programmed. The gate operators define signal relationships. Because each PLD is different, the signal names and gate operators also are different from one PLD to another. For example, a 24-pin PLD might have a signal named pin23, but a 20-pin PLD would not. The signal names for each PLD are documented by the prologic diagrams.

Look at the proLogic diagram for the 16R4. You will find it in the Logic Diagram section. Each signal is labeled with its proLogic signal name.

When you write an assignment statement, it must begin with an output signal name. Output signal names always end with the "=" character. They name gate output signals.

The remainder of the assignment is an expression comprised of any of the signal names which you want as circuit inputs. The signal names which directly feed an AND gate are written using an "&" gate operator as a separator, as in the expression

```
pin2 & pin3 & pin4
```

Similarly, signal names directly feeding an OR gate are written using the "|" gate operator as a separator.

```
pt47 | pt46                    /* device a167 */
```

When there is no signal name on a gate output, use the gate's input expression in place of the missing signal name. The expression

```
!pin19 = ( pin2 & pin3 & pin4 )
```

represents an OR gate which has an output signal name in terms of one of it's unnamed AND gate inputs.

To program pin19 as an AND instead of a NAND, write

```
! pin19 = ( ! pin2 ) | ( ! pin3 ) | ( ! pin4 )
```

where the parenthesized expressions denote the outputs of three of the AND gates feeding the OR gate. The parenthesis are not required. The expression

```
! pin19 = ! pin2   |   ! pin3 |   ! pin4
```

means the same thing.

Assignments allow a separation of the punctuation parts of a signal name from the rest using spaces, tabs or even new lines. Also, pairs of parenthesis can be used to group expressions. You can also use the symbols ″0″ and ″1″ as fixed gate inputs, as in

```
pin19 . oe = 1
```

Brackets

When you find a name like

```
[!]pin16=
```

in the prologic diagrams, the brackets indicate an optional part. In this case, either

```
pin16=
```

or

```
!pin16=
```

may be used as the output signal name.

Section 5

Symbols

Signal Names

The prologic diagrams for each Programmable Logic Device specify the signal names you are required to use in your programs. These names are the ones used in the examples in the earlier sections.

If you've looked at the Texas Instruments application briefs, or at the sample Application PLD files, you've probably noticed that it's standard practice to assign application specific names to the various signals. There are a number of good reasons for doing this:

1. PLD specifications can be complex. Appropriate mnemonics are a big help to comprehension, not only during initial PLD specification writing, but also when trying to comprehend an existing specification.
2. Certain signal names have conventional meaning, OE for Output Enable, CS for Chip Select, A7 for address line seven, and so forth. Also, some standard circuits have conventional names. To illustrate, if part of your PLD circuit is an SR latch, the inputs should be named R and S.
3. It saves extra documentation. If your PLD specification file uses the pin17.d signal name, then somewhere you must document that signal's function. On the other hand, if you call it D3 and the programmed PLD is titled "An Updown Counter", the name itself implies counter output bit three.
4. Your programmed PLD is part of a circuit. It is convenient if the names used in the PLD specification are similar to those used by your schematic capture package.

Application signal names are specified by `"define"` statements. The statement looks like this

```
define cs = pin2;          /* chip select */
```

The "define" replaces the symbol to the left of the "=" with the symbols to the right of the "=" (except the ;). This example means "wherever "cs" is found, replace it with pin2". Thus the expression

```
!pin19=cs
```

becomes

```
!pin19=pin2
```

after the define replacement.

The "define" is a powerful tool. With it you can create application specific signal names such as

```
define STATE2 = (!pin17 & pin16 & !pin15);
define STATE3 = (!pin17 & pin16 &  pin15);
define ERROR  = ! pin19;
```

so that an expression for PLD pin 19 can be written as

```
ERROR = STATE2 | STATE3
```

Define statements do not take effect until they are encountered. For this reason, it is a good idea to group them together near the start of the program.

Symbols

The proLogic compiler recognizes three kinds of symbols:

1. A group of alphanumeric characters, such as

```
cs          define
pin2        DEFINE
3725        8259_CS
```

2. A group of characters, such as

```
=         ==         &!
;         %          =!
```

   except that

```
(    {  }
```

   are always single character symbols and

```
/*
```

   always begins a comment.

3. Any characters within quotation marks

```
"this is an unusual symbol"
"/* and so is this */"
```

A comment is any text enclosed by /* and */. Since characters like newline are text, you can write very large comment blocks. Comments can be written anywhere you can use a space or a tab, that is, they are symbol separators.

Do not use any of these reserved symbols as signal names.

```
define          else            if
include         repeat          signal
state           state_diagram   test_vectors
title           truth_table
```

Do not use any symbol beginning with an underscore _ character as a signal name.

Upper and lower case letters are different. That is, "`define`" is not the same as "DEFINE", which is different from "Define". The define statement does not replace a variable's extension (a symbol after the dot). That is, if one of your variables is

```
define d = pin19;
```

the expression

```
pin17.d = d
```

becomes

```
pin17.d = pin19
```

The "`define`" statement can be used to tailor the operator symbols to your preference. proLogic uses the "`&`" for AND, "`|`" for OR and "`!`" as the negation attribute. But they can be changed to "`*`", "`+`" and "`/`" by these defines:

```
define * = & ;        define + = | ;        define / = ! ;
```

Notice that the spaces between the symbols are required. If we had written

```
define *=&;
```

proLogic would see this as the symbol "`define`" followed by the symbol "`*=&;`". If you want to change the operator symbols you also need

```
define =/ = = ! ;
define */ = & ! ;
define +/ = | ! ;
```

to be able to write a natural expression such as

```
a=/b*/c+/d
```

without being forced to put in spaces to separate symbols formed from punctuation characters, as in

```
a= /b* /c+ /d
```

The "`define`" statement rescans after it completes a replacement. For instance, if you finished inputting a program and realize that the polarity of one of the signals is wrong, you may be tempted to make a "temporary" fix with a define statement like

```
define cs = (!cs);       /* wrong */
```

This define statement may find an expression such as

```
x=cs
```

and replace it with

```
x=(!cs)
```

but then it rescans yielding

```
x=(!(!cs))
x=(!(!(!cs)))
x=(!(!(!(!cs))))
.
.
.
and so forth...
```

Statements like

```
define a=b;    define b=a;
```

present similar opportunities.

In practice, these do not occur very often. When they do, you will find that proLogic informs you what symbol it is seeing. A number of instances can be corrected by adding parentheses. As a further aid, the first part of the listing (.LST) file shows the signal specifications after everything is completed.

Section 6

Expressions

Operators

In Section 4 we wrote a 3-input AND gate as

```
! pin19 = ! pin2 | ! pin3 | ! pin4
```

to make the point that an AND gate on an active low output cell requires the consumption of three product terms. That is also the form required to map onto the proLogic Diagram. We relaxed the strict version of signal names by writing

```
!pin19=
```

as

```
! pin19 =
```

Now we are going to take the next step away from the proLogic Diagram and write

```
! pin19 = !(pin2 & pin3 & pin4)
```

When proLogic sees this kind of an expression, it uses arithmetic rules to transform it internally back into the sum-of-products form which maps to the diagram. If you compile

```
! pin19 = !(pin2 & pin3 & pin4)
```

and look at the (.LST) file, you will see that it has become

```
!pin19= !pin2 | !pin3 | !pin4
```

Expressions are evaluated according to priority. The "|" operator has a lower priority than the "&" operator. Both "|" and "&" have a lower priority than the "!" attribute. In the expression

```
!a | b & c
```

the highest priority expressions are evaluated first: first the "!"

```
(!a) & b | c
```

then the "&"

```
((!) & b) | c
```

leaving the "|" for last.

When the normal priorities work against you, there is another rule which states that parenthetical expressions have a higher priority than operators and attributes so the expression

```
!((a | b) & c)
```

becomes

```
!a & !b | !c
```

Similarly,

```
(a | b) & (c | d)
```

becomes

```
a & c | a & d | b & c | b & d
```

You can even write expressions like

```
! ( a = b & ! c )
```

which turns out to be

```
!a = !b | c
```

The assignment operator "=" has a lower priority than most, so that in the expression

```
a = b & c
```

the

```
b & c
```

expression is evaluated before it is assigned to "a". Conversely the " . " (dot) operator has a very high priority so that it applies before the "!", as in

```
! pin27 . D = 1
```

which proLogic sees as

```
! ( pin27 .d ) = 1
```

The dot operator changes upper case letters in the extension to lower case. The equality operators "==" and "!=" provide a way of writing the equivalent of the logical Exclusive NOR/OR functions. That is,

```
a == b
```

is the same as the expression

```
a & b | !a & !b
```

The "!=" operator is defined to be the inverse. Which means

```
a != b
```

is the same as

```
! ( a == b )
```

or

```
a & !b | !a & b
```

As a general rule, all operators are binary. They require an expression both before and after the operator. For example a attributes like "!" apply to a single expression.

Statements

An assignment expression such as

```
a = b & c
```

becomes a statement when it is followed by a semicolon, as in

```
a = b & c;
```

You have seen the "`define`" statement. It is also terminated by the semicolon. Syntactically, your program is a sequence of statements.

There is another kind of statement called a block which is terminated by a "`}`". The title block is the subject of the next section.

Lists and Numbers

Lists and Numbers

Lists and numbers provide an alternative way of expressing logic for bus-oriented designs. If you were to configure a P16R4 as a quad D-type flip-flop, you need to write four assignments to connect the input word to the register inputs. Rather than writing

```
d3.d = i3; d2.d = i2;  d1.d = i1; d0.d = i0;
```

you could "define" the input word and register input words as the two lists

```
define Din = (i3, i2, i1, i0);
define Dff = (d3.d, d2.d, d1.d, d0.d);
```

and write the single assignment

```
Dff = Din;
```

If your register is to have a synchronous clear to all low, you can implement it with

```
if (clear)
    Dff = 0;
  else
    Dff = Din;
```

and you can add in a synchronous set to all high using either an if-else

```
if (clear)
    Dff = 0;
  else
        if (set)
        Dff = 0xF;
      else
        Dff = Din;
```

or as an equivalent assignment

```
Dff = !clear & (Din | set);
```

An asynchronous decode on the input bus data can be specified as

```
if (Din == 5)
    isFive = 1;
```

or as the equivalent assignment expression

```
isFive = Din == 5;
```

Lists and numbers will save time, make the designs more readable, and reduce clerical errors.

Lists

The comma operator "," groups expressions together for convenience. These groups, such as

```
a, b, c
```

are called lists. Lists are usually enclosed in parentheses because the comma has a priority lower than any other operator. This is why we used parenthesis in the define of the words for the quad flip-flop.

Because lists are themselves expressions, they can be used with operators. The result of an "=" operation on the two lists

```
(a, b) = (x, y)
```

is the single list

```
(a = x), (b = y)
```

A list of assignment expressions becomes a statement by appending a semicolon. That is,

```
(a, b) = (x, y);
(a = x), (b = y);
```

and

```
{a = x;  b = y;}
```

are all equivalent. Our quad flip-flop used this form when we wrote

```
Dff = Din;
```

which after define replacement is

```
(d3.d, d2.d, d1.d, d0.d) = (i3, i2, i1, i0);
```

or

```
{d3.d = i3;  d2.d = i2;  d1.d = i1;  d0.d = i0;}
```

Numbers

A number is a symbol whose text is a sequence of digits. When writing "0" or "1", the number base does not make any difference. In fact, "0" and "1" are decimal integers. There are other notations for binary, octal and hexadecimal. A leading "0b" implies binary, "0o" octal and "0x" hexadecimal. The letters in a number can be either lower or upper case. For example, decimal 31 can be written as "0x1F" or "0X1f" in hexadecimal.

The meaning of a number depends on where it appears in the program. Depending on the context, a number can be a constant, a list of constants, an integer, or even text.

Constants

The numbers "0" and "1" can be constants. Constants reference variables with these special properties:

```
t & 0  ==  0
t | 0  ==  t
t & 1  ==  t
t | 1  ==  1
t & !t ==  0
t | !t ==  1
```

where "t" is any expression. Of course,

```
0 & t  ==  0
```

and similarly for all the others because of the commutative nature of Boolean algebra. Attributes are not applied to constants.

Constants, like other expressions, can be grouped into lists. That is, the list statement

```
(a, b, c) = (1, 0, 0);
```

yields the assignments

```
a = 1;   b = 0;   c = 0;
```

Lists of Constants

Any number except a constant can be transformed to a list of constants. For example, the decimal number "4" is transformed to the list

```
1, 0, 0
```

You may use this to write

```
(a, b, c) = 4;
```

as an equivalent, but shorter and more readable, alternative to

```
(a, b, c) = (1, 0, 0);
```

Our quad flip-flop used this form when we wrote

```
Dff = 0xF;
```

which is short for

```
Dff = (1, 1, 1, 1);
```

A binary number is converted to a list of constants which has one constant for each binary digit (bit). The binary number

```
0b00100
```

is transformed to the list

```
(0, 0, 1, 0, 0)
```

The number of bits implied by other notations is the minimum required to preserve the value; that is, leading 0s are ignored. Thus 3 is converted to the list (1, 1), but 4 becomes (1, 0, 0).

List Operations

An attribute is applied to a list by applying it to each of the grouped expressions. Applying the negation attribute, as in

```
!(a, b)
```

yields the list

```
(!a), (!b)
```

We have earlier defined the result of an "=" operation on the two lists

```
(a, b) = (x, y)
```

to be the single list

```
(a = x), (b = y)
```

Most of the other operators have identical transformations. Mentally substitute any operator in place of the "=" to determine the result. Of course, the lists can have more than two components as in

```
(a, b, c) & (x, y, z)
```

which yields

```
(a & x), (b & y) (c & z)
```

but whenever both operands are lists, they must have the same number of component. Another way to view it is that both lists must have the same number of commas.

The exceptions are the equality operators "==" and "!=" which turn the two lists into single expression. The result of

```
(a, b) == (x, y)
```

is

```
a==x & b==y
```

The result of the "!=" operator is defined to be the inverse. This means

```
(a, b) != (x, y)
```

is defined as

```
!((a, b) == (x, y))
```

which is the expression

```
a!=b | b!=y
```

The rules for the equality operators are easier to remember if you think of them as being the conditioning term in an if-else like

```
if (Din == 5)
        isFive = 1;
```

Lists and Non-Lists

When the assignment operator "=" specifies that the value of a function is a list, as in

```
x = (a, b, c)
```

the list is converted to the sum of its component expressions, yielding

```
x = a | b | c
```

The result in all other cases when one operand is a list and the other is a single expression, as in

```
(a, b, c) & x
```

is as if you had written

```
(a & x), (b & x), (c & x)
```

or

```
(a, b, c) & (x, x, x)
```

Our quad flip-flop used this form when we wrote

```
Dff = !clear & (Din | set);
```

The "| " operator has the list (Din) on the left and the single expression (set) on the right. It yields the list

```
i3 | set, i2 | set, i1 | set, i0 | set
```

The "&" operator has the single expression (`!clear`) on the left and the above list on the right. It yields the list

```
!clear & (i3 | set),
 !clear & (i2 | set)
  !clear & (i1 set)
    !clear & (i0 | set)
```

The assignment operator has two lists of the same size and yields the list statement

```
d3.d = !clear & (i3 | set),
 d2.d = !clear & (i2 | set),
  d1.d = !clear & (i1 | set),
   d2.d = !clear & (i0 | set);
```

When a list is used as the conditioning expression of an "`if-else`" statement, it is converted to the sum of its terms. For example,

```
    if (a, b) ...
```

becomes

```
    if (a | b) ...
```

In other words, it is true if any of its terms are true. This is the same conversion performed by the assignment operator "=".

Lists and Numbers

Numbers behave differently from variables in list operations. From the previous rules you may expect that

```
    (a, b) & 1
```

would yield

```
    a & 1, b & 1          /* wrong */
```

but it does not, instead you will get

```
    a & 0, b & 1
```

The reason is that numbers are considered to be a flexible form of a list. Additional ways to write the number 1 are:

```
    01  or  001  or  0001 ...
```

Because of this, when presented with

```
    (a, b) & 1
```

the compiler uses as many bits as necessary to match the size of the list. In this case, it increases the number's size from one to two bits to form

```
    (a, b) & (0, 1)
```

which yields

```
    a & 0, b & 1
```

Our quad flip-flop used numbers this way when we wrote

```
    if (Din == 5)...
```

because the number 5, which is the list (1, 0, 1), is smaller than Din. It was expanded to (0, 1, 0, 1) before the comparison.

Note that the compiler will increase the number of bits in a number, but it will not decrease them. Thus if you try to cram five signals into four, as in

```
    Dff = 16;            /* error */
```

the compiler will not let you.

Integers

The value of a number may also be considered to be a positive arithmetic integer. Integers have values in the range 0 through 0xFFFFFFFF and are arithmetic in the sense that

```
1  <  0   ==  1
22 <= 23  ==  1
47 != 4   ==  1
2  +  2   ==  4
3  -  1   ==  2
2  *  3   ==  6
7  /  2   ==  3
```

The arithmetic operators listed above plus the ">" and ">=" are all available to you. Writing

```
Dff = 2+3;
```

is the same as writing

```
Dff = 5;
```

These operators will not work on anything but numbers. They will not perform numeric logic synthesis for you.

Note that numbers are always numbers, no matter how you write them. Even though the list

```
(1, 0, 1)
```

has seven symbols, it is list of numbers and thus it is itself a number in the same way that 0b101 and 5 are numbers. Therefore,

```
Dff = (1, 0) + (1, 1);
```

is only a complicated way to obtain

```
Dff = 5;
```

In numeric contexts, a list of numbers is concatenated at the bit level so that

```
(0o0, 0x3, (0, 0o2))
```

is equivalent to

```
0b11010
```

Logic High and Logic Low

When the "!" attribute is applied to a number, it treats it as an integer and does a ones-complement arithmetic operation. Like all arithmetic operations, it is performed on 32 bits, however the "!" is the only arithmetic operation which does not recompute the number of bits in the result. Thus when the size of a number is increased to match a list, the extra bits added are 1s. For example,

```
Dff = 2;
```

is

```
Dff = (0, 0, 1, 0);
```

but

```
Dff = !2;
```

is

```
Dff = (1, 1, 0, 1);
```

When you are using lists, there is a difference between the integer 1 and the constant 1. For this reason, the definitions

```
define HIGH = (!0);
define LOW  =   0;
```

are available in the STDSYN.H header file. You may prefer to use "HIGH" when you want a constant and 1 when you want an integer. As examples,

```
Dff = HIGH;
```

is

```
Dff = (1, 1, 1, 1);
```

in the same way that

```
Dff = set;
```

is

```
Dff = (set, set, set, set);
```

but

```
Dff = 1;
```

is

```
Dff = (0, 0, 0, 1);
```

LOW is there only for consistency, there is no difference between the integer 0 and the constant 0.

Concatenation

The "\" operator makes one symbol out of two expressions. Concatenating

```
i \ 3
```

yields the symbol

```
i3
```

Using concatenation, you can write the quad flip-flop input word as

```
i \ (3, 2, 1, 0)
```

instead of

```
i3, i2, i1, i0
```

Lists of Sequential Numbers

Another way to write the list

```
5, 6, 7, 8
```

is

```
5..8
```

using the ".." operator. It also works backwards and in other number bases as in

```
0x1C..0x18
```

which is short for

```
28, 27, 26, 25, 24
```

You can mix these with other numbers to list the input pins on a P16R4:

```
    2..9, 12, 13, 18, 19
```

and is especially useful in conjunction with the concatenation operator. A shorter method to write the quad input word is

```
i \ 3..0
```

and by using the dot operator, the flip-flop word can be shortened to

```
d \ 3..0 . d
```

These two look sort of cryptic. Instead of relying on operator priority, we prefer to use explicit parenthesis and write

```
d\(3..0).d = i\(3..0);
```

The concatenation and dot operators turn numbers into text. Even when a number is really used as an integer, it can be transformed back to its text form when required. In these instances, the text chosen is that which represents a decimal number. Thus in

```
d . ( 0xA2 + 1 )
```

the arithmetic sum of the integers 0xA2 and 1 is taken yielding the integer 0xA3. This must be converted to text for the dot operator. The decimal notation for this number is 163, so the result of the expression is a reference to the variable

```
d.163
```

When lists of numbers are used in a textual context, they are left alone. (1, 0, 1) is not the same as 5.

Don't Care Constants

Binary numbers can use an x to show a don't care bit. You can also use them in octal and hex numbers to get groups of three and four. 0bx, the don't care constant, references a variable with these special properties:

```
t & 0bx == t
0bx & t == t
t | 0bx == 0bx
0bx | t == 0bx
! 0bx   == 0bx

t = 0bx == 0bx
0bx = t == 0bx
0 = 0   == 0bx
1 = 1   == 0bx
```

In the above, "t" is any expression. The statement

```
0bx;
```

is a null statement, just like

```
;
```

It is doubtful if you want to write

```
Dff = 0bx;
```

because it is a null statement, but you may write

```
Dff = 0b1x11;
```

which will not do anything to d2.d and sets the other signals logic high. Another way to write the same thing is

```
Dff & 0xB = Din & 0xB;
```

which sets all signals except d2.d to the Din values. This works because the statement

```
0 = 0;
```

is a null statement. A shorter way to write this is

```
Dff = Din | 0b0x00;
```

Its inverse,

```
Dff = Din & 0b0x00;
```

sets only d2.d to its Din value and the others to logic low while

```
Dff = Din & 0bx1xx;
```

sets d2.d to logic high and the others to the Din values.

Don't care constants are also useful as a conditioning expression, as in

```
if (Din == 0b1xxx)...
```

which you may prefer to the equivalent

```
if (Din & 0x8 == 0x8)...
```

Numbers, Lists and Truth Tables

Truth table blocks were covered in the introductory chapters. Here are some examples that were not shown because lists had not been discussed at that point:

```
truth_table { a, b, c  :  z  ;
                 0     :  1  ;
                 1     :  1  ;
                 2     :  1  ;
                 3     :  1  ;
                 4     :  1  ;
                 5     :  1  ;
                 6     :  1  ;
                 7     :  0  ;
}
```

which defines "z" as a three-input NAND gate. A 2-bit-counter could be

```
truth_table {  reset     (v1,v0).q   :  (v1,v0).d      ;
                 0          0bxx     :      0          ;
                 1             0     :      1          ;
                 1             1     :      2          ;
                 1             2     :      3          ;
                 1             3     :      0          ;
 }
```

This is a good time to note that tables do not need to be in tabular form in your program. The semicolons are the end-of-line marks, not the newlines. You may think the NAND table looks better like this:

```
truth_table {
     a, b, c: z;  0:1; 1:1; 2:1; 3:1; 4:1; 5:1; 6:1; 7:0;
 }
```

Section 8

The Title Block

A title block

```
title {   Function: Special Barrel Shifter.
          Designer: Acme Products.
          Date:     4 July 1988.
          .
          .
}
```

defines text to be copied to the JEDEC output file as documentation. The title block is a statement. It may appear almost anywhere in your program, but you can only have one. The text enclosed by the braces "{" and "}" is copied to the JEDEC output file. Because the JEDEC file format uses an "*" character to delimit the documentation, your text may not contain this character.

The semicolon is not required to terminate the title block (or any other block). Note that our style of block which has a separate line for the closing "}" and begins the text on the same line as the opening "{" may not be to your taste. Feel free to experiment to find a style that suits you.

The text is also copied to the fuse map portion of the (.LST) file.

Section 9

Truth Table Blocks

The Classic Seven-Segment Display

The following is the truth table to program a PLD as a hex decoder driver for a seven-segment display.

```
truth_table {  /*             aa
                          +---------+
                          |         |
                        bb|         |ff
                              gg
                          +---------+
                          |         |
                        cc|         |ee
                          |   dd    |
                          +---------+
               */
               q3  q2  q1  q0  :  aa  bb  cc  dd  ee  ff  gg;

  /*  0  */     0   0   0   0  :   0   0   0   0   0   0   1;
  /*  1  */     0   0   0   1  :   1   0   0   1   1   1   1;
  /*  2  */     0   0   1   0  :   0   1   0   0   1   0   0;
  /*  3  */     0   0   1   1  :   0   1   1   0   0   0   0;
  /*  4  */     0   1   0   0  :   1   0   1   1   0   0   0;
  /*  5  */     0   1   0   1  :   0   0   1   0   0   1   0;
  /*  6  */     0   1   1   0  :   0   0   0   0   0   1   0;
  /*  7  */     0   1   1   1  :   0   1   1   1   0   0   1;
  /*  8  */     1   0   0   0  :   0   0   0   0   0   0   0;
  /*  9  */     1   0   0   1  :   0   0   1   1   0   0   0;
  /*  A  */     1   0   1   0  :   0   0   0   1   0   0   0;
  /*  B  */     1   0   1   1  :   1   0   0   0   0   1   0;
  /*  C  */     1   1   0   0  :   0   0   0   0   1   1   1;
  /*  D  */     1   1   0   1  :   1   1   0   0   0   0   0;
  /*  E  */     1   1   1   0  :   0   0   0   0   1   1   0;
  /*  F  */     1   1   1   1  :   0   0   0   1   1   1   0

}
```

Each truth table block creates a block of assignment statements, but is set in tabular form to save writing time and to help visualize all cases.

The table heading line

```
        q3   q2   q1   q0   :   aa   bb   cc   dd   ee   ff   gg   ;
```

lists the input and output expressions. The input expressions are listed first and are separated from the output expressions by the colon. The semicolon ends the heading line.

Each other line of the table is a detail line. Detail lines follow the format set up by the heading line with respect to number of input and output expressions. These lines create assignment statements. If your truth table had only this single detail line

```
 /* 0 */       0    0    0    0    :    0    0    0    0    0    0    1 ;
```

then only this one assignment statement would be created:

```
    gg = q3==0 & q2==0 & q1==0 & q0==0;
```

These two detail lines

```
/*  0 */       0    0    0    0    :    0    0    0    0    0    0    1 ;
/*  1 */       0    0    0    1    :    1    0    0    1    1    1    1 ;
```

would create all these:

```
        aa = q3==0 & q2==0 & q1==0 & q0==1;
        dd = q3==0 & q2==0 & q1==0 & q0==1;
        ee = q3==0 & q2==0 & q1==0 & q0==1;
        ff = q3==0 & q2==0 & q1==0 & q0==1;
        gg = q3==0 & q2==0 & q1==0 & q0==0
           | q3==0 & q2==0 & q1==0 & q0==1;
```

And these three

```
/* 0 */      0    0    0    0    :    0    0    0    0    0    0    1   ;
/* 1 */      0    0    0    1    :    1    0    0    1    1    1    1   ;
/* 2 */      0    0    1    0    :    0    1    0    0    1    0    0   ;
```

would create these:

```
        aa = q3==0 & q2==0 & q1==0 & q0==1;
        bb = q3==0 & q2==0 & q1==1 & q0==0;
        dd = q3==0 & q2==0 & q1==0 & q0==1;
        ee = q3==0 & q2==0 & q1==0 & q0==1
           | q3==0 & q2==0 & q1==1 & q0==0;
        ff = q3==0 & q2==0 & q1==0 & q0==1;
        gg = q3==0 & q2==0 & q1==0 & q0==0
           | q3==0 & q2==0 & q1==0 & q0==1;
```

All 16 of the detail lines would produce seven assignments and the aa assignment would be the sum of four products – one for each 1 in the "aa" column. Even though the final signal specifications which show up in the listing file can be reduced to their minimum logical equivalents, a truth table is certainly a way to generate a lot of logic with minimal work on your part.

The reason inputs use the equality operator "==" is so that you can have more complicated things than 0s and 1s. In

```
        truth_table { a   b   :   z   ;
                      c   d   :   1   ;
        }
```

the assignment is

```
z = a==c & b==d;
```

Notice that we used the phrase input expression at the start of this section. This means that not only can you negate inputs, as in

```
truth_table { !a   !b  :  z  ;
               1   !d  :  1  ;
}
```

which creates the assignment

```
z = (!a == 1) & (!b == !d);
```

You can also write almost any valid expression, such as

```
truth_table { !(a & b)  c   :  z  ;
                     c   d | e :  1  ;
}
```

which is

```
z = (!(a & b)) == c & c == (d | e);
```

The only expression that you have to stay away from in the detail lines is a signal named X (in either upper or lower case). That is because we picked this signal name to mean don't care. If you need to use it, put it in parenthesis. This table:

```
truth_table { a   b  :  x  y  ;
              1   x  :  1  0  ;
              x  (x) :  0  1  ;
}
```

yields

```
x = a==1;
y = b==x;
```

The output side of the truth tables is easier because there are only three characters permitted – 0, 1 and X (or x). This will make menu selections from the output expressions in the header.

The X is again a don't care. It means that this detail line will not affect the assignment for the output. The 1 (0) specifies that the output is logic high (low) for the case defined by the input. The output characters can be written together. That is,

```
0x01
```

is the same as

```
0 X 0 1
```

Output Polarity

You should think of the polarity problem in exactly the same way that you do when you are about to write an assignment statement. Logic high (the 1s) produce assignments which are true relative to the signal specification names. If we remove the abstraction and show some real signal specification names for the 16R4 on this table:

```
    truth_table { pin2 pin3 pin4   : !pin19   ; /* 3-input NAND
*/
                    0    X    X    :    1     ;
                    1    0    X    :    1     ;
                    1    1    0    :    1     ;
                    1    1    1    :    0     ;

     }
```

it becomes clear that it is correct.

Section 10

State Diagrams

A Modulo-3 Counter

proLogic can help you implement programmable logic state machines. Other names used for this general subject are finite state machines, sequencers, sequential circuits, counters, Mealy model/machine/circuit, Moore model/machine/circuit, state diagram, ect..

If you want to create a circuit using clocked flip-flops with feedback, then you can use state blocks instead of assignments.

Consider a modulo-3 counter for an example. It uses two flip-flops as state variables. Each of the three permitted combinations of 0s and 1s in the state variables is called a state. On power up it is in one of the states and on each clock a next state is established which depends, in part, on the current state. Figure 6 illustrates a modulo-3 counter.

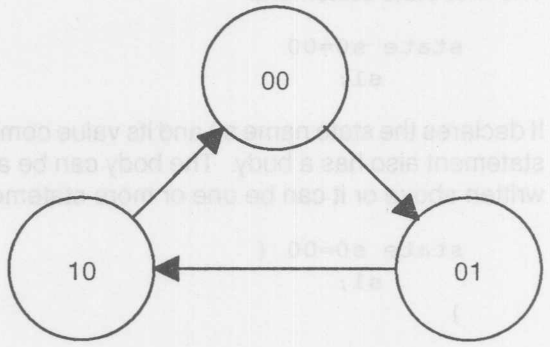


Figure 6. Modulo-3 Counter

The assignments for a D-type flip-flop implementation are determined by inspection to be

```
v1.d = v0.q;
v0.d = !v1.q & !v0.q;
```

The same counter written as a state diagram appears as follows:

```
state_diagram v1,v0 {
      /* modulo-3 counter */

      state s0=00
            s1;

      state s1=01
            s2;

      state s2=10
            s0;
}
```

A state diagram appears different from the assignments. This is because each assignment is a single statement but the state diagram has a hierarchical structure.

First while the outermost state_diagram statement.

```
state_diagram v1,v0 {
      . . .
}
```

It declares "v1" and "v0" to be state variables. Their values are defined in the body of the diagram. The body is all of the statements enclosed by the braces "{" and "}".

In our counter example, the body consists of a state statement for each of the three states.

The first state statement.

```
state s0=00
      s1;
```

It declares the state name s0 and its value combination (v1.q,v0.q)=(0,0). The state statement also has a body. The body can be a single statement without braces, as written above or it can be one or more statements enclosed by braces, as in

```
state s0=00 {
      s1;
}
```

The transition statement

```
s1;
```

consists of a state name, as it appears in one of the state statements of this state diagram, followed by a semicolon. It specifies what state occurs on the next clock.

IF-ELSE

The "`if-else`" statement is used to condition other statements. The general form is

```
if (expression)

          statement-1                /* then part */

else

          statement-2                /* else part */
```

where the else part is optional.

To show its use, let us make our modulo-3 counter stay low (in state s0) until triggered by an external signal (count) as shown in Figure 7.

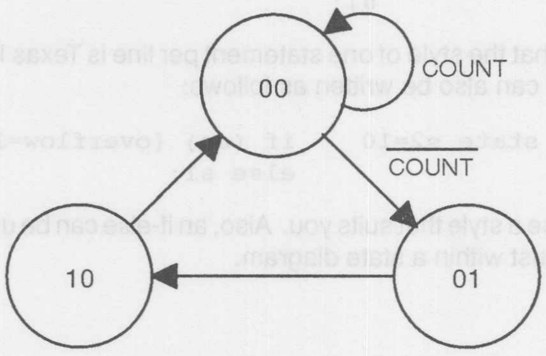


Figure 7. Modulo-3 Counter

This is coded

```
state_diagram v1,v0 {
          state s0=00
                    if (count)
                              s0;
                    else
                              s1;
          state s1=01
                    s2;
          state s2=10
                    s0;
}
```

The "`if-else`" statement conditions its then part with the expression in parenthesis and its else part with the expression's complement. Because its then part (or else part) can be another "`if-else`" statement or a block (one or more statements enclosed by braces "`{`" and "`}`"), things can become as complex as necessary.

The if-else and state statements can also condition assignments.

You can add an output to state s2 by writing

```
state s2=10 {
        overflow=1;
        s0;
}
```

or make it an up-down counter with

```
state s2=10
        if (up) {
                overflow=1;
                s0;
        }
        else
                s1;
```

Note that the style of one statement per line is Texas Instruments style. State statements can also be written as follows:

```
state s2=10     if (up) {overflow=1; s0;}
                else s1;
```

Choose a style that suits you. Also, an if-else can be used anywhere in your program – not just within a state diagram.

Global Transitions

Extend our modulo-3 counter so instead of being triggered to count, it stays in state 00 unless enabled.

```
state_diagram v1,v0 {
        state s0=00
                if (count)
                        s0;
                else
                        s1;
        state s1=01
                if (count)
                        s0;
                else
                        s2;
        state s2=10
                s0;
}
```

State machines often have some condition which applies to every state. These global transitions can be factored out and written before the first state.

The diagram

```
    state_diagram v1,v0 {
            if (count)
                    s0;
            state s0=00
                    s1;
            state s1=01
                    s2;
            state s2=10
                    s0;
    }
```

looks more like what it really is – a modulo-3 counter with an active high reset.

Illegal States

If the modulo-3 counter (shown in Figure 8) is implemented on flip-flops which power up to a random state the counter may be in trouble. It counts

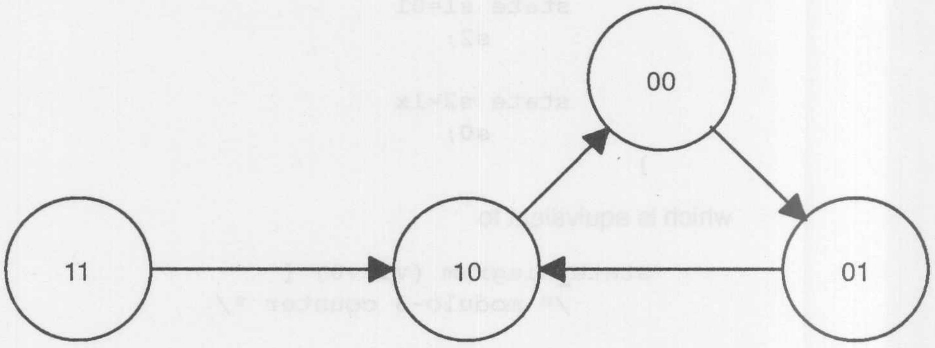


Figure 8. Modulo-3 Counter

This may be acceptable. If it is not, you can correct it by adding an additional

```
    state s3=11     /* illegal */
            s0;
```

If your machine is something other than a counter you can also specify that a state has don't cares for some, but not for all, of its values. Instead of adding a new state, the existing state s2 could be redefined as

```
    state s2=1x
            s0;
```

The "x", which can be either upper or lower case, means that either a 10 or a 11 identifies state s2.

State Variables

The state variables declared by the "state_diagram" statement must name a flip-flop with internal feedback. The extension may be omitted. For example

```
state_diagram pin17,pin16 {
          . . .
}
```

declares that the pin17.q signal holds the most significant state value for a 16R4 implementation. On this PLD, proLogic synthesizes assignment statements for pin17.d= and pin16.d=.

You can precede a state variable with a "!" which inverts all logic for that flip-flop. The modulo-3 counter example, if written for a 16R4 with active low outputs would be

```
state_diagram !v1.q, !v0.q {
        /* p16r4 modulo-3 counter */

        state s0=00
                s1;

        state s1=01
                s2;

        state s2=1x
                s0;
}
```

which is equivalent to

```
state_diagram (v1,v0) {
        /* modulo-3 counter */

        state s0=11
                s1;

        state s1=10
                s2;

        state s2=0x
                s0;
}
```

Conditioning

The statements which form the body of the state block are conditioned by the state. The following is a state which is the highest state of a modulo-5 counter

```
state s4=1XX {
    Y=1;
    s0;
}
```

Because the expression "`Y=1`" is contained within the body of state s4, it is conditioned by s4. This means the resulting expression is extended to be true only when state variable v2 (the most significant variable) is logic high and yields

```
Y = v2.q;
```

If Y is a combinatorial output, it becomes high shortly after clocking a 1 into v2. If Y is a registered output, it becomes high on the next clock which simultaneously causes v2 to go low as it transitions to state s0. What happens to Y after that depends on what is written in the state s0 block. Conversely, if you wanted register Y to go high on the clock of 1 into v2 to enter state s4, Y would need to be set up in the prior state. For an up-counter,

```
state s3=000 {
    Y=1;
    s4;
}
```

would do it, but in an up-down counter, both preceding states are required to set up Y.

```
state s0=000
    if (up)
        s1;
    else {
        Y=1;
        s4;
    }

state s3=011
    if (up) {
        Y=1;
        s4;
    }
    else
        s2;
```

Default Transitions

An unconditioned transition statement is called a default transition. In

```
state s0=00
    s1;
```

the s1 is a default transition because it is not conditioned by an "`if-else`" statement. The state

```
state s0=00
    if (count)
        s0;
    else
        s1;
```

could also have been written

```
state s0=00 {
    if (count)
        s0;
    s1;
}
```

because the default transition specifies the state change when none of the other transitions apply.

States which have no default transition remain in their current state when none of the other transitions apply. Thus

```
state s0=00
    if (!count)
        s1;
```

also holds in s0 while count is logic high.

Each state block can have only one default transition statement.

Section 11

Test Vector Blocks

Test vectors are used by device programmers or logic simulators to verify that the logic functions defined for a PLD are correct. These vectors describe the inputs to the programmed PLD and specify the outputs expected after applying each set of inputs.

Each test_vectors block defines a sequence of vectors to be applied to a set of test condition variables. These blocks are written in tabular form with semicolons separating each line of the table.

```
test_vectors {

        pin3    !pin1    pin18   pin19;
         1        1        H       L;
         1        0        L       L;
         0        0        L       H;
}
```

The first line of the table

```
        pin3    !pin1    pin18   pin19;
```

names the test condition variables. In your program you probably have some "define" statements to make the names more meaningful. Because test conditions are applied to the pins of the device, not internal signals, you do not necessarily need to adhere strictly to the proLogic diagram signal names. Minor variations will perform as expected.

Each other line of the table is a test vector. The first vector

```
         1        1        H       L;
```

specifies the test conditions 1, 1, H and L to be applied respectively to the test condition variables pin3, !pin1, pin18 and pin19. Spaces are not required. You can write

```
         1        1        H       H ;
```

or

```
        11HH;
```

as long as there is one test condition for each test condition variable. Lower case letters can be used if you prefer. This table lists the allowable test conditions.

0	-	drive input LOW	-	1
1	-	drive input HIGH	-	0
2..9	-	drive input to supervoltage #2-9	-	2..9
C	-	drive input LOW-HIGH-LOW	-	K
F	-	float input or output	-	F
H	-	test output HIGH	-	L
K	-	drive input HIGH-LOW-HIGH	-	C
L	-	test output LOW	-	H
N	-	power pins and outputs not tested	-	N
P	-	preload registers	-	P
X	-	output not tested, input default level	-	X
Z	-	test input or output for high impedance	-	Z

The last column lists the inverse condition which is used when a test condition variable is negated. The block

```
test_vectors { !pin1;
               1;
               0;
               N        }
```

is equivalent to

```
test_vectors { pin1;
               0;
               1;
               N        }
```

There is a special kind of a test vector which is used to create one or more JEDEC buried register vectors. This one begins with the symbol internal and is followed by as many 0s, 1s, Ls and Hs as there are internal registers in the PLD.

Note:

If you are new to PLD design, be aware that we have left out some very important practical considerations. When a device programmer executes preload vectors, or vectors which have supervoltages, it may apply high voltages to pins. Refer to the data sheets for your exact PLD. The device programmer manual also provides important cautions to observe when testing.

The Repeat Block

A repeat block, written

```
    repeat N {
        . . .
    }
```

can be used anywhere in your program. Its function is to create N copies of the symbols contained in the body. The following is a sample application.

```
test_vectors {
     /* 4-bit counter partial test */

     pin1      pin17 pin16 pin15 pin14;

      P          0     0     0     0;    /* preload */

   repeat 15 {
      C          X     X     X     X;    /* clock 15 times */
   }

      0          H     H     H     H;    /* OK if all high */

      C          L     L     L     L;    /* wraparound */

}
```

The } which terminates the repeat block was placed on the line after

```
      C          X     X     X     X;    /* clock 15 times */
```

but could also have been placed on the same line after the semicolon.

Repeat blocks are not statements and can be used almost anywhere. Nesting is permitted, such as

```
   repeat 15 { C repeat 4 { X }; }
```

Section 12

Include Files

Exactly one

```
include file-name;
```

which identifies the Architecture Description file (.LXA) for the target device must be present somewhere in each program. The ″`file-name`″ is a symbol string, such as

```
P16R4.LXA
```

or

```
\prologic\p22v10.lxa
```

When you do not specify a file extension of .LXA, the ″`include`″ statement has a different function. In this case, the include and its symbols are replaced by the contents of the identified file.

You may want to use ″`include`″ files to define commonly used symbols like

```
define LOW = 0;
```

to save time and assure consistency over all of your PLD programs.

Even though a semicolon terminates the file-name, the ″`include`″ is not a statement. Like the repeat block, it can be used almost anywhere. Included files may themselves have ″`include`″. A repeat block may contain ″`include`″ (and vice versa).

When no file extension is supplied, the default is .H (for header). But if your file has spaces as an extension, make this clear.

```
include myfile. ;
include myfile.″   ″;
include ″myfile.   ″;
```

Header Files

The proLogic compiler provides a library of header files (.H). These provide extra levels of information about each device to make it easier to write a program. You may think of them as a subroutine library. Header files answer questions like:

```
how many pins?
is register preload available?
what pins have registers?
how do states map to D flip-flops?
to SR flip-flops?
does   if (c) x=a;   mean   x= (c & a)?
does it mean    x = (c & a) | (!c & x.q)?
does the compiler drive output enables when you don't?
what about logic minimization?
.
.
.
```

The only logical operators which the compiler recognizes are the AND and OR. Header files may offer other operators such as XOR.

When you select a header file such as

```
include p16r4;
```

at the beginning of your program, you will not need an

```
include p16r4.lxa;
```

because the header file will bring the Architecture Description in as part of itself.

If your preference is for as little help as possible, use an `"include"` statement such as

```
include p16r4.lxa;
```

at the start of your program. This requires you to be very explicit when writing the program. Most of the higher level statements can not be used in this mode. We recommend you `"include"` the STDSYN.H file in addition to the LXA file. It has `"define"` statements which will relax the operator symbols to make the expressions more natural.

The proLogic Header Library

The header files extend the basic proLogic compiler capabilities. While the use of the library is not required, we recommend that you review the tools for possible use in your applications.

Acknowledgement

The mnemonics used to declare signal TYPES are copyright Altera Corporation and are used by permission. We would like to express our appreciation to the Altera Corporation not only for permitting the use of material under copyright, but also for allowing the use of the previously unpublished information contained in the proLogic Diagrams.

Library Selection Guide

Some functions are implemented by selecting the appropriate header for a particular device.

These header files have been written to ease PLD designs by allowing a simpler method of defining pins, counters, and combinatorial logic. The following table shows, by function and device, the proper library header for each of these functions. All files have an implicit (.H) extension.

FUNCTION	P330	P630	P1830
++ −−	DCNTR	DCNTR TNCTR	DCNTR TNCTR
state diagram synthesis	DFF	DFF TFF	DFF TFF
logic minimization	MINPALS	MINPALS	MINPALS

File: AT.H

Function: Replaces application signal names with proLogic Diagram signal names using the "@" operator.

Usage: The "@" operator replaces all occurrences of its left operand with a signal name of pinN, where N is the numeric right operand.

This header file provides the PIN attribute which permits free-standing pin declaration statements.

The "@" operator can be used in conjunction with a TYPE header file. As such, it is automatically included in main header files for the EP330, EP630, and EP1830.

Related: device-name.TYP

Required: A device-name.H device description header

Examples: The statement

```
!(out1 @ 19).d = a @ 2 & out1.q;
```

becomes

```
!pin19.d = pin2 & pin19.q;
```

An equivalent program can be written using the "PIN" statement if you prefer to declare the variables separately.

```
            PIN out1 @ 19;
            PIN a @ 2;
            !out1.d = a & out1.q;
```

Factored alternatives to the two "PIN" statements above are:

```
            PIN (out1, a) @ (19, 2);
```

or

```
            PIN (out1 @ 19
                 a    @ 2  );
```

Refer to the type header for additional examples of the factored form of variable declarations.

When used with a TYPE header file, the appropriate type mnemonic is used instead of the PIN attribute as in this program:

```
title { Application:     74374 OCTAL D FLIP-FLOP
}
include p330;           /* the device description header */
include p330.typ;       /* the TYPE header */
include at;             /* the @ operator */
include tvector;        /* vector compile-time checks */

define Din = (in\7..0);
define Dout = (out\7..0);

INP (Din,oe) @ (9..2,11);
RONF Dout @ 12..19;

Dout.d = Din;
Dout.oe= !oe;

test_vectors {  pin1 oe in7 in6 in5 in4 in3 in2 in1 in0
                out7 out6 out5 out4 out3 out2 out1 out0 ;

        0 1 11111111 ZZZZZZZZ; /* not enabled */
        0 0 11111111 LLLLLLLL; /* power up low */
        C 0 11111111 HHHHHHHH; /* clock */
        0 0 01011011 HHHHHHHH; /* hold */
        C 0 01011011 LHLHHLHH; /* clock */
        0 1 11111111 ZZZZZZZZ; /* not enabled */
}
```

File: COMPARE.H

Function: Comparator Logic Synthesis using the following operators:

`<`	Less Than
`<=`	Less Than or Equal
`>`	Greater Than
`>=`	Greater Than or Equal

Usage: These operators are used the same way as the proLogic Compiler equality operators "==" (Equal to) and "!=" (Not equal to). Like the equality operators, they yield a single-valued result when used with list operands.

These operators are typically used to compare a list of variables to a number. Caution should be observed when both operands are lists of variables as even a relatively small number of variables synthesize large Boolean expressions.

Required: No additional header files are required, however it may be desirable to include the appropriate minimization header for the device.

Example:

```
title { SIMPLE DECADE UP-COUNTER }

include p630;                /* device description header */
include p630.typ;            /* the TYPE header */
include at;                  /* the @ operator */
include compare;             /* the <= operator */
include tcntr;               /* the ++ operator */
include minpal8;             /* logic minimization */

define count = (q\3..0);
TOTF count @ 15..18;

if (count.q < 9)
        ++count.t;
  else
        count.t = 0;
```

File: DCNTR.H

Function: D Flip-flop Up/Down Binary Counter Logic Synthesis using the following operators:

`++`	Count Up
`--`	Count Down

Usage: These operators must precede a single operand in a free-standing separate statement. The operand must be the D input signal of a D Flip-flop or a list of such signals.

The .q extension is used in logic synthesis. If using global macrocells from different quadrants, this must be considered in counter designs (see appendix A).

Related: TCNTR.H

Required: No additional header files are required, however it may be desirable to include the appropriate minimization header for the device.

Example:

```
title { BINARY FREE-RUNNING DOWN-COUNTER
}

include p630;            /* device description header */
include p630.typ;        /* the TYPE header */
include at;              /* the @ operator */
include dcntr;           /* the -- operator */
include minpal8;         /* logic minimization */

RORF (d2, d1, d0) @ 17..19;
   --(d2, d1, d0).d;
```

File: MINPAL.H

Function: Logic Minimization for sum-of-products architectures without programmable polarity.

Usage: The minimization function may be requested by the compiler for statements which are contained within a state diagram, or those which are contained within an "`if-else`" statement. Minimization may also be requested by other headers. It may be requested directly by preceding an assignment statement with the "`MINIMIZE`" attribute.

Do not use the "`MINIMIZE`" attribute within a state diagram or within an "`if-else`" statement. Assignment statements within these blocks are automatically minimized.

This header may be used instead of a device-specific minimization header to force simple sum-of-product minimization regardless of the device architecture.

Related: MINPAL8.H, MINPLA.H

Required: none

Example:

```
include minpal;
MINIMIZE out.d = a & b | a;
```

File: MINPAL8.H

Function: Logic Minimization for devices with eight product terms and programmable polarity on the gate input (sum-of-products output). If sum-of-products minimization yields a function with more than eight product terms, a minimization of the product-of-sum form of the function is attempted.

Usage: The minimization function may be requested by the compiler for statements which are contained within a state diagram, or those which are contained within an "`if-else`" statement. Minimization may also be requested by other headers. It may be requested directly by you by preceding an assignment statement with the "`MINIMIZE`" attribute.

Do not use the "`MINIMIZE`" attribute within a state diagram or within an "`if-else`" statement. Assignment statements within these blocks are automatically minimized.

Related: MINPAL.H, MINPLA.H

Required: none

Example: see MINPAL.H

File: TCNTR.H

Function: T Flip-flop Up/Down Binary Counter Logic Synthesis using the following operators:

```
++      Count Up

--      Count Down
```

Usage: These operators must precede a single operand in a free-standing separate statement. The operand must be the T input signal of a T Flip-flop or a list of such signals.

The .q extension is used in logic synthesis. If using global macrocells from different quadrants, this must be considered in counter designs (see appendix A).

Related: DCNTR.H

Required: No additional header files are required, however it may be desirable to include the appropriate minimization header for the device.

Example:

```
title { BINARY FREE-RUNNING DOWN-COUNTER
}

include p630;          /* device description header */
include p630.typ;      /* the TYPE header */
include at;            /* the @ operator */
include tcntr;         /* the -- operator */
include minpal8;       /* logic minimization */

TOTF (d2, d1, d0) @ 17..19;

--(d2, d1, d0).t;
```

File: Device.TYP

Function: SIGNAL TYPING The source file (.PLD) that you create to specify your circuit needs to:

```
- name the device

- define the application names

- specify each circuit output as a function
  of its input
```

Device Naming

A device is selected by writing, for example

```
include p1830;          /* device description header file */
include p1830.typ;      /* the TYPE header file */
include at;             /* use @ to assign pin names */
```

near the start of your program. The first statement names the device description header file (P1830.H), and is required in all programs using signal typing. There is a device description header for each device. There is also a TYPE header for each device. The one for the example device is named by the second statement. The third statement is the header which permits the use of the `"@"` operator style of defining application signal names. It is generic to any device. The addition of the second and third `"include"` statement is unnecessary since they are included in the main header for the EP330, EP630, and EP1830.

Application Signals

Next, each of the input and output signal names are declared, given a type, and assigned a device resource. The four statements:

```
CONF nand3 @ 68;
INP a @ 59;
INP b @ 58;
INP c @ 57;
```

declare that nand3 is TYPE **CONF** (Combinatorial Output, No Feedback) with an assignment to the macrocell on pin 68. Signals a, b, and c are TYPE **INP** (Input) and are assigned to pins 59 through 57. Select the TYPE names from Figure 9 on the next page.

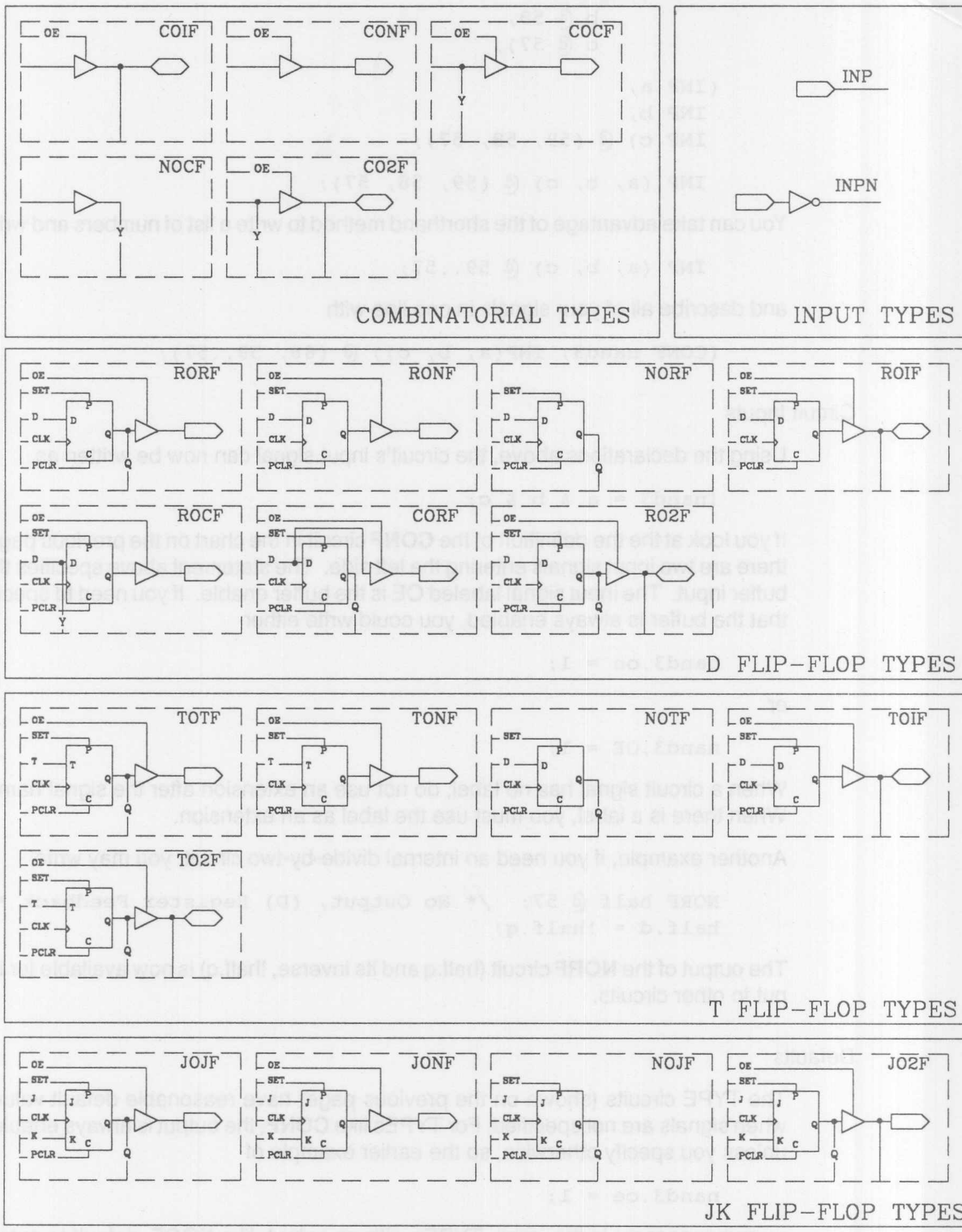


Figure 9. Application Signal TYPE Names

Time can be saved when writing declarations by factoring. Factoring permits a single type name or single "@" operator to be applied to more than one signal name. Additional ways to write the three "INP" statements are:

```
      INP (a @ 59,
           b @ 58,
           c @ 57);

     (INP a,
      INP b,
      INP c) @ (59, 58, 57);

      INP (a, b, c) @ (59, 58, 57);
```

You can take advantage of the shorthand method to write a list of numbers and write

```
      INP (a, b, c) @ 59..57;
```

and describe all of your signals in one line with

```
     (CONF nand3, INP(a, b, c)) @ (68, 59..57);
```

Circuit Inputs

Using the declarations above, the circuit's input signal can now be written as

```
     !nand3 = a & b & c;
```

If you look at the the definition of the **CONF** circuit in the chart on the previous page, there are two input signals entering the left side. The statement above specifies the buffer input. The input signal labeled OE is the buffer enable. If you need to specify that the buffer is always enabled, you could write either

```
     nand3.oe = 1;
```

or

```
     nand3.OE = 1;
```

When a circuit signal has no label, do not use an extension after the signal name. When there is a label, you must use the label as an extension.

Another example, if you need an internal divide-by-two circuit, you may write

```
     NORF half @ 57;   /* No Output, (D) Register Feedback */
     half.d = !half.q;
```

The output of the **NORF** circuit (half.q and its inverse, !half.q) is now available for input to other circuits.

Defaults

The TYPE circuits (shown on the previous page) have reasonable default values when signals are not specified. For TYPEs like **CONF**, the output is always enabled unless you specify otherwise, so the earlier example of

```
     nand3.oe = 1;
```

was unnecessary in this case. TYPEs with no output, like **NOCF**, default to output not enabled. Preclear (PCLR) and Set (SET) default to inactive, while clocks (CLK) are the same as the default specified by the proLogic diagram for the device.

Dual Feedback

The nand3 used an input c which was assigned to pin 57. The divide-by-two specified a buried register also on pin 58. Since pin 57 on the P1830 is a global macrocell, there is no conflict. It is not permissible to have both an **INP** TYPE and a **RORF** TYPE on the same pin. The way to declare this is to omit the **INP** for c and redefine half as

```
RO2F half @ 57;  /* (D) Register Output, Dual Feedback */
```

Then override the **RO2F** default output enable with

```
half.oe = 0;
```

and write the NAND gate as

```
!nand3 = a & b & half;
```

Active Low Inputs

Another way to write a 3-input NAND gate is

```
nand3 = !a | !b | !c;
```

An alternative is to declare an active LOW input by writing

```
INPN (a, b, c) @ (59, 58, 57);
```

or

```
! INP (a, b, c) @ (59, 58, 57);
```

and then changing the assignment to

```
nand3 = a | b | c;
```

You can declare selected signals to be active LOW by factoring only the INP, as in

```
INP (a, !b, c) @ 59..57;  /* b is active LOW */
```

or

```
! INP (!a, b, !c) @ 59..57;  /* b is active LOW */
```

Active Low Outputs

You can specify output buffer inversion by negating the TYPE declaration, as in

```
! CONF nand3 @ 68;
nand3 = a & b & c;
```

When you negate a TYPE declaration, mentally add a bubble to the output buffer of the circuit.

Usage: RESTRICTION: Output Buffer Inversion is currently restricted to those devices which provide actual programmable polarity at the output buffer.

```
RESTRICTION:  SET is not implemented on any type.
```

Example:

```
RORF d3;
d3.set = 1:  /* not implemented */
```

Related: AT.TYP

Required: A device-name.H device description header. The IF.H header is required for JK flip-flops.

File: XOR.H

Function: Implementation of the Exclusive Or Boolean Function using the "%" operator.

Usage: The "%" operator is used exactly the same as the compiler's logical "|" operator. It has a lower priority than the logical "|", so that the expression

```
    a = c | a & b % d;
```

is evaluated as if it were written

```
    a = ((c | (a & b)) % d;
```

For most operands, the XOR synthesizes the equivalent function expressed in AND and OR operators. That is,

```
    a % b
```

is expanded to

```
    !a & !b | a & b
```

As a special case, when a function of the form

```
    pinNN.d = y % q;
```

or

```
    pinNN.d = q % y;
```

exists, where "pinNN" is a macrocell which can be be REGISTER BLOCK PROGRAMMED as a T Flip-flop, it generates the equivalent signal statement.

Related: none

Required: A device-name.H device description header is required if REGISTER BLOCK PROGRAMMING synthesis is desired, otherwise no other header files are required.

Example:

```
title { 74280 9-BIT ODD/EVEN PARITY GENERATOR/CHECKER }

include p330;                   /* device description header   */
include p330.typ;               /* the type header             */
include at;                     /* the @ operator              */
include minpal8;                /* logic minimization          */
include xor;                    /* the % operator              */

INP         in\8..0 @ 1..9;
CONF (   odd @ 19,              /* output */
            even @ 18);
COIF (   sum1 @ 17,       /* internal NOCF */
            sum2 @ 16);
```

```
sum1 = in0 % in1 % in2 % in3;
sum2 = in4 % in5 % in6 % in7;
odd  = in8 % sum1 % sum2;
!even= in8 % sum1 % sum2;
```

Section 13

Signal Specifications

Although proLogic has the ability to manipulate, optimize, and even synthesize Boolean equations. Everything is put into the form which will map onto the proLogic Diagram signal names and gate operators. This form is called a signal specification.

In the example

```
!pin19 = pin2 & pin3 & pin4
```

The signal specification is

```
"!pin19=" pin2 & pin3 & pin4
```

Quotes are used to write the seven-character symbol

```
!pin19=
```

because that is the exact name on the proLogic diagram. Names such as

```
!pin2
```

would also need to be quoted.

The rules for writing a signal specification are as follows:

1. A signal specification always begins with an output signal name. These names always end with an "=" (equal) character.

2. An input signal name must follow the output signal name. These names never end with an "=" character.

3. After the first input signal name, more input signal names can be listed if they are separated by gate operators.

Gate Operators

The prologic diagrams implicitly use these operators:

`&`	`Logical AND Gate`
`|`	`Logical OR Gate`
`%`	`Logical Exclusive OR Gate`

Other operators unique to the PLD may be specified by individual diagrams.

The characters

`!`	`Logical Negation`
`=`	`Assignment`
`.`	`Dot`

are not gate operators. They are required to "spell" some signal names.

Gate Operators Function

There is always a conceptual "current gate". The importance of the current gate concept is that it provides the environment for interpreting the meaning of the input signal names. Input names always identify signals with respect to the current gate.

The current gate becomes important in multi-level logic. The output name usually identifies a hierarchical logic structure of one, two, or more levels. For example, in a 16R4, the output name pin19.oe= identifies a one level product term. The output name !pin19= identifies a two level sum of product terms. The current gate is always one of the gates at the lowest level of the hierarchy. In both of these examples, the current gates are AND gates.

The initial current gate is identified by the output name. It is always the first (or topmost) of all of the possible current gates. The operator which identifies the current gate (in these examples, the "`&`"), has no effect on the current gate. It serves only as a separator between input names. An operator which identifies a gate at a higher level in the PLD logic structure (in these examples, the "`|`") changes the current gate to be the next of all possible current gates.

In a one level structure such as the 16R4 pin19.oe=, there is only one possible current gate – the 32-input AND which drives the output enable for pin 19. It is the initial current gate identified by the pin19.oe= output name. In the signal specification for this signal only "`&`" operators may appear.

In the two level structure identified by the !pin19= output name, there are seven AND gates at the lowest level. The output name identifies the first of these as the current gate. All input names identify connections to this AND gate until an "`|`" operator is encountered. After the "`|`", all input names identify connections to the second AND gate, until the next "`|`" when they apply to the third, and so forth.

0 And 1

There are two special signal names which apply to the current gate of all devices. They are "1", which means the output of the current gate is logically true, and "0", which means the the output of the current gate is logically false. Since these names apply to the gate output, no other input names may apply to that gate. For example,

```
"pin19.oe=" 1 & pin2
```

is improper.

The "0" and "1" signal names used in the signal specification are different from the "0" and "1" symbols used in expressions. In an expression, they are gate input signals. That is,

```
( pin19.oe = 1 & pin2 )
```

is transformed into

```
( pin19.oe = pin2 )
```

In the first matrix of a 16R4 there are 256 programmable cells. These two signal specifications cause only the last cell to remain connected, while the first 255 are disconnected (programmed).

```
"pin19.oe=" 1

"!pin19="    1 | 1 | 1 | 1 | 1 | 1 |  "!pin12"
```

The Signal Statement

Most PLD programs for most devices can be expressed entirely in terms of boolean equations using logical AND, OR and NOT. That's why the assignment statement such as

```
a = b & !c;
```

requires you to do the minimum amount of typing on your keyboard. But some PLDs have XOR gates or signature words.

For these instances the signal statement always provides the ability to program at the signal specification level. Another way of writing

```
a = b & !c;
```

is

```
signal "a=" b & "!c";
```

In this statement, everything between signal and the semicolon must be a signal specification.

You also have the option of letting proLogic do most of the work for you. In the signal statement, anything enclosed by parenthesis is an expression. Thus

```
signal "a=" b & (!c);
signal "a=" (b & !c);
signal (a = b & !c);
```

are all equivalent.

To make it easier, you can also write the `"="`, `"."`, and `"!"` symbols separately.

proLogic will translate

```
signal a = ! b . c;
```

into

```
signal "a=" "!b.c" ;
```

because it knows the rules for forming a signal name.

X330 Translator

This program accepts a 20 pin PLD JEDEC file and translate it into a EP330 JEDEC code which will yield a pin for pin replacement device.

Implementation:

Each fuse in the source (simple PLD) JEDEC file is remapped into the equivalent fuse location for an EP330. For example:

- In a 10H8 'fuse 0' connects 'pin 2' with the first AND term in the 'pin 19' output. In a EP330 'fuse 39' connects 'pin 2' to the first AND term in the 'pin 19' output. This continues for all source JEDEC file fuses.
- Any unused terms in the EP330 are left as is.
- If a source device pin does not have an output enable, all output enable fuses for that pin are programmed in the output JEDEC file. Therefore the output enable is always enabled.
- For all source device registered pins, the output enable fuses are all programmed except for one connection to 'pin 11'.
- Unused pin connections for valid rows are programmed. For example:

  In a 16R8 all pins fed to the array except for pins 1 and 11. The fuses for these in the EP330 output file will be programmed. This is to avoid a false signal feeding back from a 'unused' pin input or feedback.

Operation:

The program is used by typing:

```
X330 <SOURCE> <TARGET> <ENTER>
```

The program then responds with a menu of source device types enumerated 1 through 17. The number entered corresponds to the input file device type. The program then translates the input source file assuming the number entered is correct.

Devices Supported:

10H8	10L8
12H6	12L6
14H4	14L4
16H2	16L2
16H8	16L8
16HD8	16LD8
16RP4	16R4
6RP6	16R6
6RP8	16R8
16P8	

Test Vector Notes:

Test vectors are transferred over into the JEDEC file when the translator is executed. Two comments need mentioning:

1. Although the IMPACT-X™ technology PLDs (–5, –7, –10) and the EP330 both have power-up logic 0 registers, the IMPACT-X™ PLDs will initially have a high level on the output pins due to the inverting buffer between the register and the pin itself. The EP330 will have a low level on the output pin due to the absence of that inverting buffer. This situation can be avoided by always having initialization vectors in the test vector so both the EP330 and IMPACT-X™ technology PLDs are in the same state at the beginning test vector set.

2. If IMPACT™ technology PLDs (–15, –25) are in use, remember that the registers in these devices power-up logic 1, and the proLogic Simulator assumes a logic 0 power-up registers for these devices. Again initialization test vectors can be used to avoid this situation.

Appendix A

Using the EP1830 as a 4 Hexadecimal Digit Counter and Display

This appendix shows how the TI EPLD, EP1830, can be used to implement a 4-digit hexadecimal counter and display design. Written in proLogic, it shows how header files can be incorporated into a design to avoid writing lengthy equations.

The design is broken down into different modules for ease of analysis. Each module is presented as implemented both by the proLogic software and the actual architecture of the device, see figure A-1. The block diagram of the application is shown in figure A-2.

The Counter

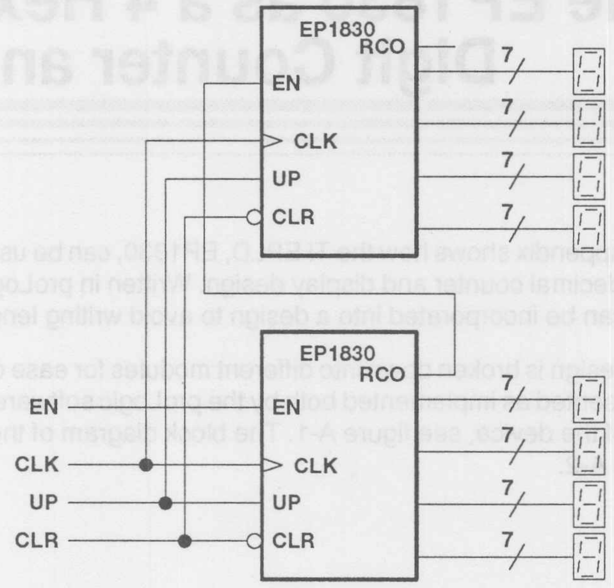
This is a resettable 16-bit binary up/down counter, using the global macrocells of the EP1830 to spread out the count value of the counter among the device's quadrants. The **TO2F** TYPE (T Flip-Flop) is used as the macrocell of choice since counter designs can be implemented with a few product terms. The control of the counter is handled through an "if-then-else" language structure. The reset of the T flip-flop is accomplished by setting its T-input to the output of that flip-flop logically ANDed with the assertion of the reset signal (CLEAR).

The actual counting up and down of the counter is greatly simplified through the use of the "TCNTR.H" header. This header file allows the designer to indicate which input stimuli will cause the counter to count up or to count down. In this example an enable count signal (EN) is used to facilitate cascading these devices when wired to the ripple carry out (RCO) of the previous device.

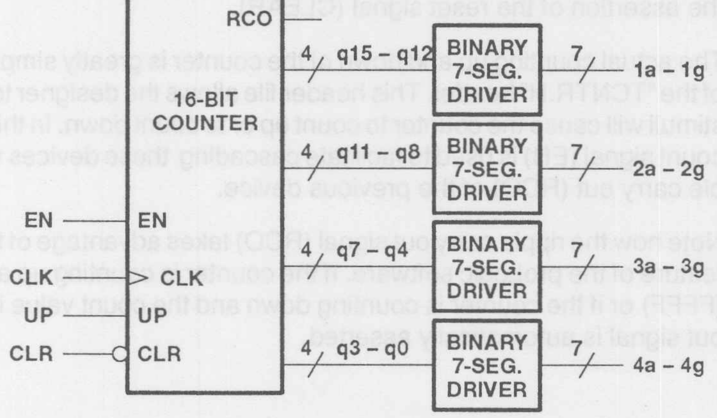
Note how the ripple carry out signal (RCO) takes advantage of the lists and numbers feature of the proLogic software. If the counter is counting up and the count value is (FFFF) or if the counter is counting down and the count value is (0), the ripple carry out signal is automatically asserted.

The Display

This is a 7-segment display with the display driver outputs being a modified version of the example in this proLogic manual, For ease of text flow the same truth table is repeated four times with the input and output signal names changed to handle each 4 binary bit of the counter.



A-1. Typical Application



A-2. Block Diagram

```
title { Device:      EP1830

        Application:  16-BIT UP/DOWN COUNTER WITH SYNCHRONOUS CLEAR
                      AND 7-SEGMENT DISPLAY DRIVER OUTPUTS

        Author:       Ivan Zellner
                      Texas Instruments FPL Applications
                      8330 LBJ Freeway MS 8317
                      Dallas TX 75243
                      214-997-5644   Phone
                      214-997-5650   Fax
                      214-997-5665   FPL BBS
    }
   include p1830;
   include tcntr;                 /* increment and decrement functions    */

define count = (q\15..0);
define count4 = (q\3..0);
define count3 = (q\7..4);
define count2 = (q\11..8);
define count1 = (q\15..12);
define 7seq  = (1a, 1b, 1c, 1d, 1e, 1f, 1g,
               2a, 2b, 2c, 2d, 2e, 2f, 2g,
               3a, 3b, 3c, 3d, 3e, 3f, 3g,
               4a, 4b, 4c, 4d, 4e, 4f, 4g);

TO2F count @ (10..13,23..26,44..47,57..60);   /* T flip-flop registers */

INP (!clear, up, en  )   @ (14, 15, 20  );
CONF 7seq @ (2..8,27..33,37..43,61..67);
CONF   rco @ 68;

     if (clear)
               count.t = count.q;
     else if (up & en)
               { ++count4.t;
                 if (count4 == 15)                               ++count3.t;
                 if ((count4 == 15) & (count3 == 15))            ++count2.t;
                 if ((count4 == 15) & (count3 == 15) & (count2==15)) ++count1.t;
                                                                      }

     else if (!up & en)
               { --count4.t;
                 if (count4 == 0 )                               --count3.t;
                 if ((count4 == 0 ) & (count3 == 0 ))            --count2.t;
                 if ((count4 == 0 ) & (count3 == 0 ) & (count2==0 )) --count1.t;
                                                                      }

     if (((count    == 0xFFFF) & up & en) | ((count    == 0) & !up & en))
        rco=1;
```

```
truth_table { q15.q  q14.q  q13.q  q12.q : 1a 1b 1c 1d 1e 1f 1g ;
              0      0      0      0     :  1  1  1  1  1  1  0 ;
              0      0      0      1     :  0  0  0  0  1  1  0 ;
              0      0      1      0     :  1  0  1  1  0  1  1 ;
              0      0      1      1     :  1  0  0  1  1  1  1 ;
              0      1      0      0     :  0  1  0  0  1  1  1 ;
              0      1      0      1     :  1  1  0  1  1  0  1 ;
              0      1      1      0     :  1  1  1  1  1  0  1 ;
              0      1      1      1     :  1  0  0  0  1  1  0 ;
              1      0      0      0     :  1  1  1  1  1  1  1 ;
              1      0      0      1     :  1  1  0  0  1  1  1 ;
              1      0      1      0     :  1  1  1  0  1  1  1 ;
              1      0      1      1     :  0  1  1  1  1  0  1 ;
              1      1      0      0     :  1  1  1  1  0  0  0 ;
              1      1      0      1     :  0  0  1  1  1  1  1 ;
              1      1      1      0     :  1  1  1  1  0  0  1 ;
              1      1      1      1     :  1  1  1  0  0  0  1 ;


              }
truth_table { q11.q  q10.q  q9.q   q8.q  : 2a 2b 2c 2d 2e 2f 2g ;
              0      0      0      0     :  1  1  1  1  1  1  0 ;
              0      0      0      1     :  0  0  0  0  1  1  0 ;
              0      0      1      0     :  1  0  1  1  0  1  1 ;
              0      0      1      1     :  1  0  0  1  1  1  1 ;
              0      1      0      0     :  0  1  0  0  1  1  1 ;
              0      1      0      1     :  1  1  0  1  1  0  1 ;
              0      1      1      0     :  1  1  1  1  1  0  1 ;
              0      1      1      1     :  1  0  0  0  1  1  0 ;
              1      0      0      0     :  1  1  1  1  1  1  1 ;
              1      0      0      1     :  1  1  0  0  1  1  1 ;
              1      0      1      0     :  1  1  1  0  1  1  1 ;
              1      0      1      1     :  0  1  1  1  1  0  1 ;
              1      1      0      0     :  1  1  1  1  0  0  0 ;
              1      1      0      1     :  0  0  1  1  1  1  1 ;
              1      1      1      0     :  1  1  1  1  0  0  1 ;
              1      1      1      1     :  1  1  1  0  0  0  1 ;

              }

truth_table { q7.q   q6.q   q5.q   q4.q  : 3a 3b 3c 3d 3e 3f 3g ;
              0      0      0      0     :  1  1  1  1  1  1  0 ;
              0      0      0      1     :  0  0  0  0  1  1  0 ;
              0      0      1      0     :  1  0  1  1  0  1  1 ;
              0      0      1      1     :  1  0  0  1  1  1  1 ;
              0      1      0      0     :  0  1  0  0  1  1  1 ;
              0      1      0      1     :  1  1  0  1  1  0  1 ;
              0      1      1      0     :  1  1  1  1  1  0  1 ;
              0      1      1      1     :  1  0  0  0  1  1  0 ;
              1      0      0      0     :  1  1  1  1  1  1  1 ;
              1      0      0      1     :  1  1  0  0  1  1  1 ;
              1      0      1      0     :  1  1  1  0  1  1  1 ;
              1      0      1      1     :  0  1  1  1  1  0  1 ;
              1      1      0      0     :  1  1  1  1  0  0  0 ;
              1      1      0      1     :  0  0  1  1  1  1  1 ;
              1      1      1      0     :  1  1  1  1  0  0  1 ;
              1      1      1      1     :  1  1  1  0  0  0  1 ;

              }
```

```
truth_table { q3.q   q2.q   q1.q    q0.q : 4a 4b 4c 4d 4e 4f 4g ;
                0      0      0       0   :  1  1  1  1  1  1  0 ;
                0      0      0       1   :  0  0  0  0  1  1  0 ;
                0      0      1       0   :  1  0  1  1  0  1  1 ;
                0      0      1       1   :  1  0  0  1  1  1  1 ;
                0      1      0       0   :  0  1  0  0  1  1  1 ;
                0      1      0       1   :  1  1  0  1  1  0  1 ;
                0      1      1       0   :  1  1  1  1  1  0  1 ;
                0      1      1       1   :  1  0  0  0  1  1  0 ;
                1      0      0       0   :  1  1  1  1  1  1  1 ;
                1      0      0       1   :  1  1  0  0  1  1  1 ;
                1      0      1       0   :  1  1  1  0  1  1  1 ;
                1      0      1       1   :  0  1  1  1  1  0  1 ;
                1      1      0       0   :  1  1  1  1  0  0  0 ;
                1      1      0       1   :  0  0  1  1  1  1  1 ;
                1      1      1       0   :  1  1  1  1  0  0  1 ;
                1      1      1       1   :  1  1  1  0  0  0  1 ;

                }
```

Programmable Frequency Divider

A Single Chip Solution and A Multiple Chip Solution

Introduction

The purpose of this appendix is to provide a comparison between the EP630 and the TIBPAL22V10 architecture capabilities and also illustrate some of the latest Programmable Logic Device (PLD) design tools offered by Texas Instruments. A brief description of the EP630 and '22V10 device architectures will be given followed by two unique implementation examples. The first with one EP630 and the second using two TIBPAL22V10 devices. Finally, a comparison will be made between the two designs.

Device Architectures

Although the EP630 and TIBPAL22V10 are both available in the same package types, the device architectures are considerably different. Both devices utilize a programmable "AND", fixed "OR" type of architecture common to all programmable logic devices. The EP630 has eight programmable "AND" terms, called Product Terms, per output while the '22V10 can utilize up to 16 product terms on some outputs. The main difference between the two devices which makes the EP630 better suited for this application is the number of registers available in each device. The EP630 has 16 registers available while the '22V10 has only 10, one per macrocell. A complete description of the device architectures and macrocell capabilities can be found in the device data sheets.

Design Methodology

In the first example, design development will be accomplished using the proLogic software. This example will consist of a 12-bit counter and the control logic which allows any one of the 12 counter bits to be routed to a single output thus creating a programmable frequency divider.

Solution 1: Single Chip Frequency Divider Implemented Using EP630

Figure 1 shows a functional block diagram which represents the EP630 design. Three 4-bit frequency dividers were used to implement the 12-bit division. Notice that the divide-by-16 bit output of each device was fed to the clock input of the next. By connecting these outputs to each successive stage, a 2^1-to-2^{12} frequency divider can be implemented.

The next level is a multiplexing level which routes the appropriate divided output to the final output, FDIV. This task is accomplished using an 8-to-1 and a 4-to-1 multiplexer. Extra control logic is necessary to provide the final level of multiplexing to the output. The input signals A, B, C, and D are used to select the desired frequency division. The table below illustrates how the device will function:

INPUT				**FREQUENCY DIVISION**
A	**B**	**C**	**D**	**DIVIDE BY**
0	0	0	0	2^1
0	0	0	1	2^2
	•			•
	•			•
	•			•
1	0	1	0	2^{11}
1	0	1	1	2^{12}

Finally, the design is compiled using the proLogic software and a standard JEDEC file is produced which can then be programmed into a single EP630.

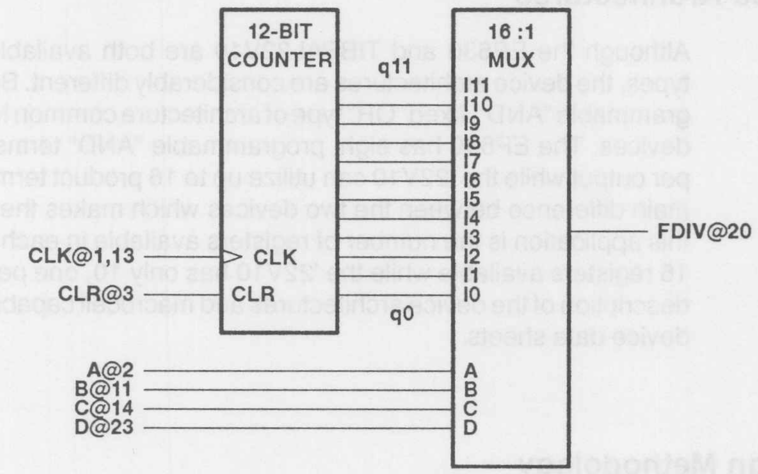


Figure 1. Schematic of the EP630 Solution

```
title { Device:          EP630

        Application:     12-BIT FREQUENCY DIVIDER WITH SYNCHRONOUS CLEAR

        Author:          Ivan Zellner
                         Texas Instruments FPL Applications
                         8330 LBJ Freeway MS 8317
                         Dallas TX 75243
                         214-997-5644  Phone
                         214-997-5650  Fax
                         214-997-5650  FPL BBS
      }
   include p630;
   include tcntr;                    /* increment and decrement functions */

define count = (q\11..0);

TOTF count @ (4..10, 15..19);  /* T flip-flop registers */
INP (clear, A, B, C, D )       /* active HIGH */
  @ (3, 2, 11, 14, 23);

CONF FDIV @ 20;
COIF FDIVA @ 21;
COIF FDIVB @ 22;

FDIV.oe = 1;
FDIVA.oe = 1;
FDIVB.oe = 1;

if (clear)
    count.t = count.q;
  else
             ++count.t;

FDIV = FDIVA | FDIVB;

FDIVA = q0.q & !A & !B & !C & !D |
        q1.q & !A & !B & !C &  D |
        q2.q & !A & !B &  C & !D |
        q3.q & !A & !B &  C &  D |
        q4.q & !A &  B & !C & !D |
        q5.q & !A &  B & !C &  D ;

FDIVB = q6.q & !A &  B &  C & !D |
        q7.q & !A &  B &  C &  D |
        q8.q &  A & !B & !C & !D |
        q9.q &  A & !B & !C &  D |
        q10.q & A & !B &  C & !D |
        q11.q & A & !B &  C &  D ;
```

Solution 2: Multiple Chip Frequency Divider Implemented Using Two TIBPAL22V10 Devices

In this next example two TIBPAL22V10 devices will be used to implement the frequency divider. Program Listings 1 and 2 show the proLogic codes developed for this application. The code shown in Listing 1 is for device A which provides the 10 lower bits of the 12 bit counter. First, the pin assignments are made which include a clock, clear, and 10 counter outputs. Next, the D input to each register is specified using

Boolean equations followed by output enable and polarity configurations. Finally, test vectors are specified which allows the code to be simulated before actually programming a device.

The next proLogic code shown in Listing 2 is for the B device which provides the two most significant bits of the 12-bit counter as well as the multiplexing control for the outputs. The clock input for the B device is driven by the Q9 output from the A device. It can be seen from the Boolean equations in Listing B that 12 product terms are needed to implement the FDIV output. Each of these "AND" terms is referred to as a Product Term. This is why the '22V10 was chosen for this application since as many as 16 product terms are available with this architecture where most PLDs have a maximum of 8 product terms per output.

Diagrams of both solution 1 and 2 including device pinouts are shown in Figure 2. A complete copy of the proLogic software and manual explaining syntax, design entry methods, device support and simulation can be obtained from your local TI sales office or by calling the TI Programmable Logic applications hotline at (214) 997–5666.

Listing 1: Device A

```
title {  Device:            TIBPAL22V10
      Application:         12 Bit Programmable Frequency Divider: Device A
         Source:           Kyle Newman  Texas Instruments        9/89       }

include p22v10;           /* specify that target device is TIBPAL22V10 */

define  CLK = pin1 ;      /* define input pins */
define  CLR = pin2 ;

define  Q0 = pin14 ;      /* define output pins */
define  Q1 = pin15 ;
define  Q2 = pin16 ;
define  Q3 = pin17 ;
define  Q4 = pin18 ;
define  Q5 = pin19 ;
define  Q6 = pin20 ;
define  Q7 = pin23 ;
define  Q8 = pin22 ;
define  Q9 = pin21 ;

/* define equations to implement lower 10 bits of counter */

  Q0.d = !(Q0.q) & !CLR ;

  Q1.d = (!Q0.q & Q1.q | Q0.q & !Q1.q) & !CLR ;

  Q2.d = (!Q0.q & Q2.q | !Q1.q & Q2.q | Q0.q & Q1.q & !Q2.q) & !CLR ;

  Q3.d = (!Q0.q & Q3.q
        | !Q1.q & Q3.q
        | !Q2.q & Q3.q
        | Q0.q & Q1.q & Q2.q & !Q3.q) & !CLR ;

  Q4.d = (!Q0.q & Q4.q
        | !Q1.q & Q4.q
        | !Q2.q & Q4.q
        | !Q3.q & Q4.q
        | Q0.q & Q1.q & Q2.q & Q3.q & !Q4.q) & !CLR ;

  Q5.d = (!Q0.q & Q5.q
        |!Q1.q & Q5.q
        |!Q2.q & Q5.q
        |!Q3.q & Q5.q
        |!Q4.q & Q5.q
        | Q0.q & Q1.q & Q2.q & Q3.q & Q4.q & !Q5.q) & !CLR;

  Q6.d = (!Q0.q & Q6.q
        |!Q1.q & Q6.q
        |!Q2.q & Q6.q
        |!Q4.q & Q6.q
        |!Q3.q & Q6.q
        |!Q5.q & Q6.q
        |Q0.q & Q1.q & Q2.q & Q3.q & Q4.q & Q5.q & !Q6.q) & !CLR;


Q7.d = (!Q0.q & Q7.q
        | !Q1.q & Q7.q
        | !Q2.q & Q7.q
        | !Q3.q & Q7.q
```

```
          | !Q4.q & Q7.q
          | !Q5.q & Q7.q
          | !Q6.q & Q7.q
          | Q0.q & Q1.q & Q2.q & Q3.q & Q4.q & Q5.q & Q6.q & !Q7.q) & !CLR;
  Q8.d = (!Q0.q & Q8.q
          | !Q1.q & Q8.q
          | !Q2.q & Q8.q
          | !Q4.q & Q8.q
          | !Q3.q & Q8.q
          | !Q5.q & Q8.q
          | !Q6.q & Q8.q
          | !Q7.q & Q8.q
          | Q0.q & Q1.q & Q2.q & Q3.q & Q4.q & Q5.q & Q6.q & Q7.q & !Q8.q) &
!CLR;

 Q9.d = (!Q0.q & Q9.q
          | !Q1.q & Q9.q
          | !Q2.q & Q9.q
          | !Q3.q & Q9.q
          | !Q4.q & Q9.q
          | !Q5.q & Q9.q
          | !Q6.q & Q9.q
          | !Q7.q & Q9.q
          | !Q8.q & Q9.q
 | Q0.q & Q1.q & Q2.q & Q3.q & Q4.q & Q5.q & Q6.q & Q7.q & Q8.q & !Q9.q) &
!CLR;

/*  permanently enable all counter outputs  */
Q0.oe = 1;  Q1.oe = 1;  Q2.oe = 1;  Q3.oe = 1;  Q4.oe = 1;
Q5.oe = 1;  Q6.oe = 1;  Q7.oe = 1;  Q8.oe = 1;  Q9.oe = 1;

/*  define outputs as active high  */
Q0 = q;  Q1 = q;  Q2 = q;  Q3 = q;  Q4 = q;
Q5 = q;  Q6 = q;  Q7 = q;  Q8 = q;  Q9 = q;

/* define some test vectors to verify that counter is working properly  */
test_vectors { /* CLK   CLR       Q3    Q2    Q1    Q0    */
                  pin1  pin2     pin17  pin16  pin15  pin14;
                  C      1          L      L     L     L    ;  /* RESET */
                  C      1          L      L     L     L    ;  /* RESET */
                  C      0          L      L     L     H    ;  /*   1   */
                  C      0          L      L     H     L    ;  /*   2   */
                  C      0          L      L     H     H    ;  /*   3   */
                  C      0          L      H     L     L    ;  /*   4   */
                  C      0          L      H     L     H    ;  /*   5   */
                  C      0          L      H     H     L    ;  /*   6   */
                  C      0          L      H     H     H    ;  /*   7   */
                  C      0          H      L     L     L    ;  /*   8   */
                  C      0          H      L     L     H    ;  /*   9   */
                  C      0          H      L     H     L    ;  /*  10   */
                  C      0          H      L     H     H    ;  /*  11   */
                  C      0          H      H     L     L    ;  /*  12   */
                  C      0          H      H     L     H    ;  /*  13   */
                  C      0          H      H     H     L    ;  /*  14   */
                  C      0          H      H     H     H    ;  /*  15   */
                  C      0          L      L     L     L    ;  /*   0   */
                  C      1          L      L     L     L    ;  /* RESET */

 }
```

Listing 2: Device B

```
title {  Device:            TIBPAL22V10

         Application:       12 Bit Programmable Frequency Divider:  Device B
                            2 MSB and Multiplexing Control
         Source:            Kyle Newman  Texas Instruments      9/89       }

include p22v10;             /* specify that target device is TIBPAL22V10 */

define  CLK = pin1 ;        /* clock input is from Q9 on device A        */
define  CLR = pin2 ;        /* clear goes to pin 2 on both devices       */

define  A = pin3 ;          /* select inputs for multiplexing outputs    */
define  B = pin4 ;          /*   Select Input          Q Output          */
define  C = pin5 ;          /*    ABCD = 0            divide by 2        */
define  D = pin6 ;          /*           1                  4            */
                            /*           2                  8 ..etc      */

define  Q0 = pin7   ;       /* define counter inputs from device A       */
define  Q1 = pin8   ;
define  Q2 = pin9   ;
define  Q3 = pin10  ;
define  Q4 = pin11  ;
define  Q5 = pin13  ;
define  Q6 = pin14  ;
define  Q7 = pin15  ;
define  Q8 = pin16  ;
define  Q9 = pin17  ;

define  Q10 = pin19 ;        /*  define 2 MSB of 12 bit counter  */
define  Q11 = pin20 ;

define  FDIV = pin18 ;        /*  divided output  */

 /*  define equations to implement 2 MSB of counter */
  Q10.d = !(Q10.q) & !CLR ;

  Q11.d = (!Q10.q & Q11.q | Q10.q & !Q11.q) & !CLR ;

 /*  define equations to control multiplexing of outputs   */

  FDIV =   Q0    & !A & !B & !C & !D
        |  Q1    & !A & !B & !C &  D
        |  Q2    & !A & !B &  C & !D
        |  Q3    & !A & !B &  C &  D
        |  Q4    & !A &  B & !C & !D
        |  Q5    & !A &  B & !C &  D
        |  Q6    & !A &  B &  C & !D
        |  Q7    & !A &  B &  C &  D
        |  Q8    &  A & !B & !C & !D
        |  Q9    &  A & !B & !C &  D
        | Q10.q &  A & !B &  C & !D   /*".q" was used on these terms      */
        | Q11.q &  A & !B &  C &  D;  /* because they are internal feedbacks */

 /*  permanently enable all outputs  */
  Q10.oe = 1;  Q11.oe = 1;  FDIV.oe = 1;

 /*  define outputs as active high  */
  Q10 = q;  Q11 = q;
```

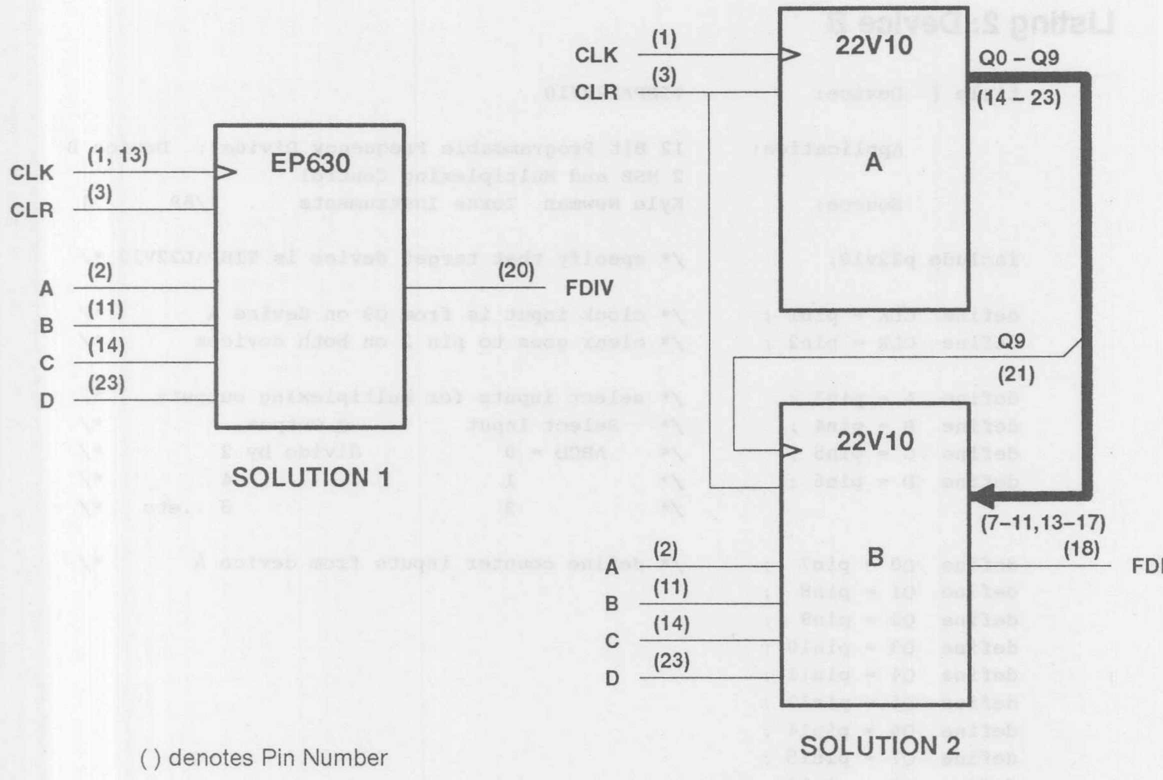


Figure 2. EP630 vs TIBPAL22V10 Solution

Conclusion

Following is a brief comparison of the two design implementations discussed in the applications note.

	SOLUTION 1	SOLUTION 2
DEVICE TYPE	EP630	TIBPAL22V10
PACKAGE	24 PIN DIP	24 PIN DIP
MAX SUPPLY CURRENT	60 mA	180 mA
PROPAGATION DELAY	20 ns	15 ns – 25 ns
MAX CLOCK FREQUENCY	50 MHz	28.5 – 40 MHz
PRICE PER PACKAGE	1.7X	X
NUMBER OF DEVICES	1	2

From this comparison, it can be seen that the EP630 is a better solution for this application. Not all applications will yield these results between a EP630 and a '22V10. However in register intensive applications such as the frequency divider, an EP630 can offer considerable savings in power consumption and package count while providing enhanced performance at a comparable price.

Appendix C

A Designer's Guide to the TIBPSG507

Robert K. Breuninger and Loren Schiele
with Contributions by
Joshua K. Peprah

Contents

List of Illustrations

INTRODUCTION

The term PSG stands for Programmable Sequence Generator. The PSG is the newest member of the programmable logic family. It combines the powerful benefits of programmable array logic (PALs) with the specialized world of Field Programmable Logic Sequencers (FPLSs).

Applications such as waveform generators, state machines, timers, and simple logic reduction are all possible with a PSG. By utilizing the built-in binary counter, the PSG is capable of generating complex timing controllers. In short, the PSG offers the system designer an extremely powerful building block.

The purpose of this application report is to describe the functional operation of the PSG507 and demonstrate how it can be applied in real-world applications. Three design examples that highlight the features and flexibility of the PSG will be discussed.

FUNCTIONAL DESCRIPTION

Figure 1 shows the architecture of the PSG507. Major features include 13 inputs, eight programmable registered or nonregistered outputs, eight S/R state registers, and a 6-bit binary counter with control logic. The clock input is fuse-programmable for selection of positive or negative edge triggering.

The binary counter, state registers, and output cells are synchronously clocked by the fuse-programmable clock input. The clock polarity fuse selects either positive or negative edge triggering. Negative edge triggering is selected by blowing the clock polarity fuse. Leaving this fuse intact selects positive edge triggering.

Each output cell on the PSG can be configured for registered or nonregistered operation through the output multiplexer fuse. Nonregistered operation is selected by blowing the output multiplexer fuse. Leaving this fuse intact selects registered operation.

The PSG507 has 13 inputs, each providing a true and complement input to the AND array. Pin 17 functions as either an input and/or an output enable. Blowing the output enable fuse lets pin 17 function as an output enable but does not disconnect pin 17 from the input array. When the output enable fuse is intact, pin 17 functions only as an input with the outputs being permanently enabled.

The 6-bit binary counter is controlled by a synchronous clear and a count/hold function. Each control function has a nonregistered and registered option. When either SCLR0 or SCLR1 is taken active high, the counter resets to zero on the next active clock edge. When either $\overline{\text{CNT}}$/HLD0 or $\overline{\text{CNT}}$/HLD1 is taken active high, the counter is held at the present count and is not allowed to advance on each active clock edge. The SCLR feature overrides the $\overline{\text{CNT}}$/HLD feature when both functions are simultaneously active high. The functional benefit of both these features will be further clarified in the examples shown later in this appliction report.

The eight internal state registers feed back into the AND array. These registers can be used to store input data, to keep track of binary count sequences, or they can be used as output registers when connected to a nonregistered output cell. The state registers differ from the output registers in that they feed back into the input array. They can also be used to override an operating sequence such as demonstrated in the designer notes located at the end of this application report. By using extra state registers, the 6-bit counter can be expanded as shown in the second example. Other uses of the internal state registers will become apparent upon reading the examples shown.

THEORY OF OPERATION

The PSG architecture is capable of operating in many different modes. When comparing the operation of a PSG to a PAL, the outputs in both devices can be configured as an AND/OR function of the inputs. One major difference between a PSG and a PAL is that a programmable OR array is used in the PSG. This allows a selected number of AND terms to be connected to each output as compared to a fixed number of AND terms assigned to each output on a PAL. The programmable OR array is the more efficient in that it lets the user assign the exact number of AND terms to each output as required by the application.

Another major difference between the PAL architecture and that of a PSG is that the output cells on a PSG are not fed back into the input array. Typically, output feedback is used for building a counter or for holding state information. Since the architecture of the PSG already includes state registers and a binary counter, the requirement for output feedback is eliminated in most applications. This is a benefit to the user because valuable output cells and AND terms are not wasted when generating these functions.

When a Field Programmable Logic Sequencer is compared to a PSG, the most obvious difference is the addition of a binary counter. Most state machine designs can be simplified by referencing all or part of each sequence to a binary count. This technique is highlighted in the third example shown in this application note. A comparison will also reveal that the output cells on a PSG can be configured

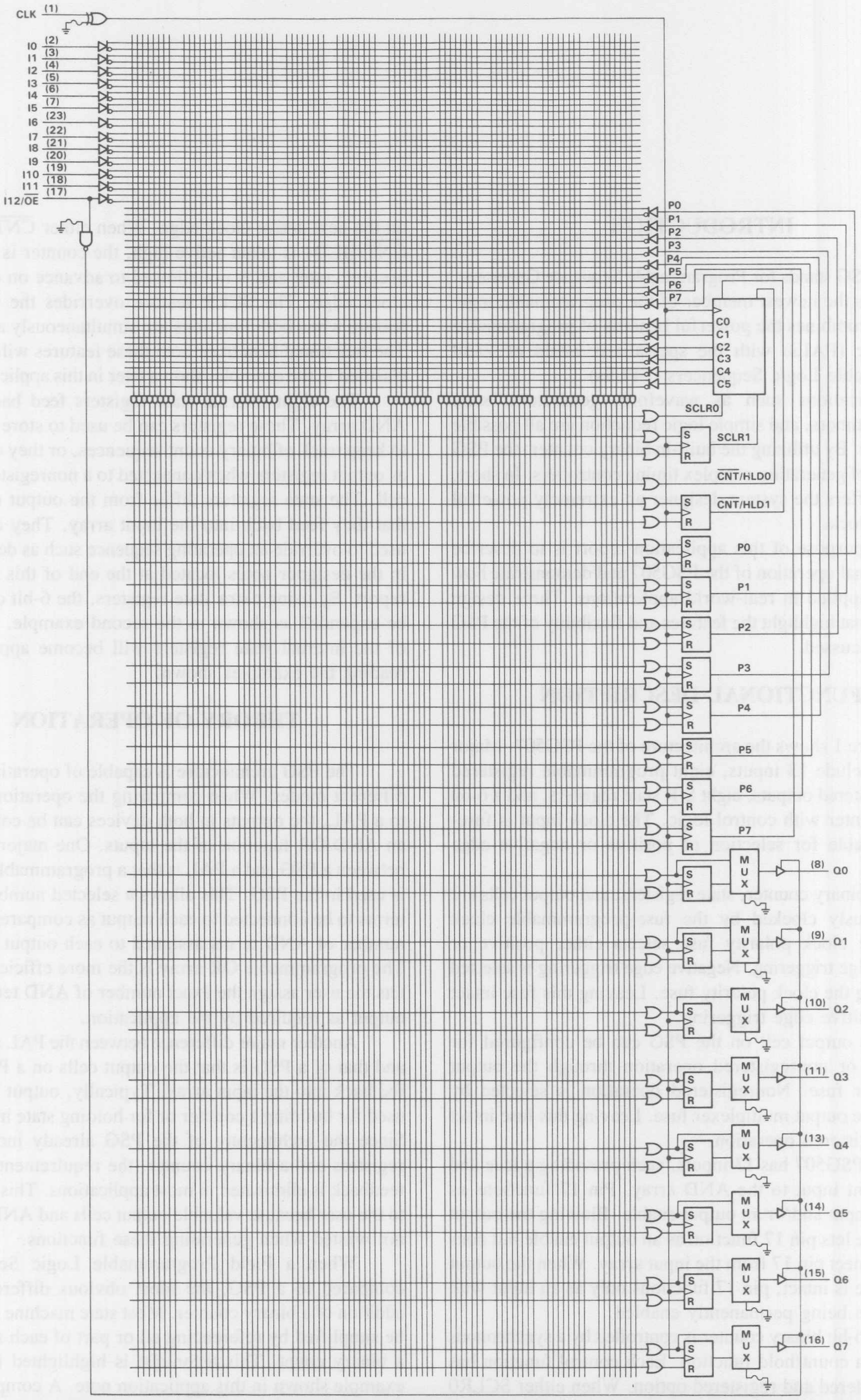


Figure 1. PSG507 Architecture

for nonregistered operation. This permits the outputs to be directly fed from the counter, AND/OR array, or state registers. Example 1 highlights this feature.

In short, the outputs of a PSG can be controlled by any or all of the following conditions:

- Present state of the inputs
- Present state of the binary counter
- Present state of the state holding registers

The key to understanding state machine design when using a PSG is to realize that different states can be assigned for each sequence. In other words, the assigned state determines which sequence is in operation. The length of each sequence is controlled by the SCLR function. Once the count sequence has been programmed to the desired length, each output can be easily decoded from the present state of the binary counter. The user will soon discover that complex state machines are easily developed when using this technique. This technique is demonstrated in Example 3.

Example 1: Waveform Generator

The first example demonstrates a design for a simple clock generator used for driving a microprocessor operating at 5 MHz (required duty cycle of 33.5% high, 66.5% low). In addition to the 5 MHz system clock (SYS CLK), a reference clock (REF CLK) operating at 15 MHz (50% duty cycle) and a peripheral clock (PCLK) operating at 2.5 MHz (50% duty cycle) are required for other timing controllers and peripherals throughout the system. Both clocks must be in close phase with the SYS CLK to guarantee synchronous operation within the system.

The above example demonstrates one of the many uses of the binary counter in the PSG. State registers are not used in this particular application, only the binary counter and three outputs. A 30 MHz clock, typically generated from a crystal, is used for driving the binary counter of the PSG. The three generated clock signals are decoded from the binary count. The unused inputs and outputs are still available for other sequential or combinational applications.

Figure 2 shows the timing diagram for the above application. For reference, a decimal count has been assigned to the master clock (PSG CLK) of the PSG. As shown in the timing diagram, at count 11 (1011_2) the sequence is repeated. By using the SCLR0 function, a logic equation can be defined to reset the counter at count 11. This concept is demonstrated in Figure 3.

With the binary counter programmed to clear at 11, it is a simple matter to decode the outputs from the binary count. With the REF CLK equal to the inverse of binary count zero (C0), REF CLK can be directly generated from the binary counter. A product term is required to connect C0 to the output cell. The output register is bypassed by blowing the output multiplexer fuse. Figure 4 shows how C0 can be connected.

SYS CLK and PCLK are decoded from the present state of the binary counter through the S/R outputs. Since the S/R register holds its present state until changed, product terms have to be used only during output transitions. For example, when the binary counter reaches one, a product term is used to reset the SYS CLK on the next clock transition. Below is a summary of the product terms required to control SYS CLK and PCLK. Note that the output transitions are set up in the previous clock cycle. Also note that only one product term is used regardless of how many output terms switch. This is demonstrated at count 5 and count 11. Figure 4 also shows how SYS CLK and PCLK are connected.

CNT 1:	Reset SYS CLK
CNT 5:	Set SYS CLK, reset PCLK
CNT 7:	Reset SYS CLK
CNT 11:	Set SYS CLK, set PCLK

This simple application demonstrates the basic concept of building a waveform generator using the PSG. This concept will be expanded further in Example 3 when a memory timing controller is developed. The basic rules for building a waveform generator are summarized below.

- Program the counter to reset to zero after the desired count length is reached.
- Generate the logic equations to control the outputs from the present state of the binary counter.

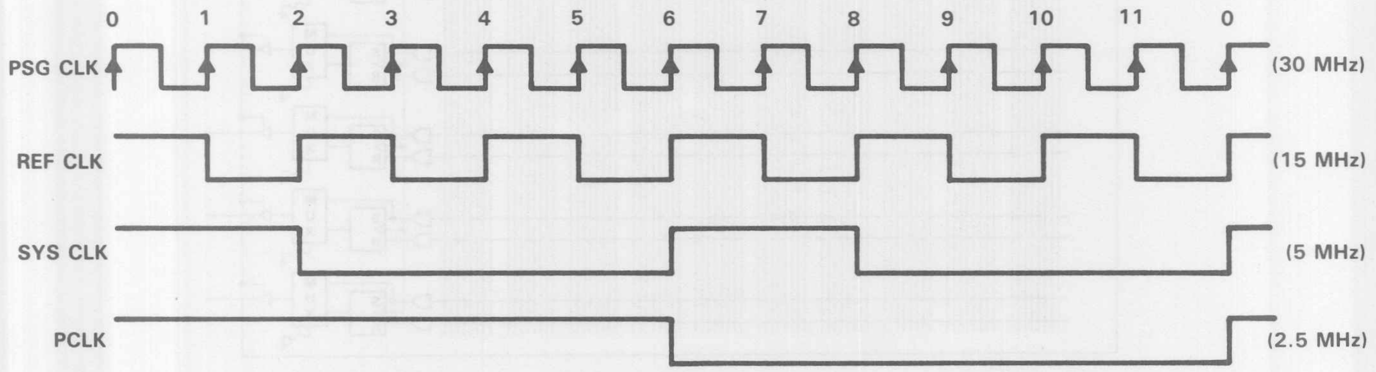


Figure 2. Clock Generator Timing Requirements (Example 1 — Waveform Generator)

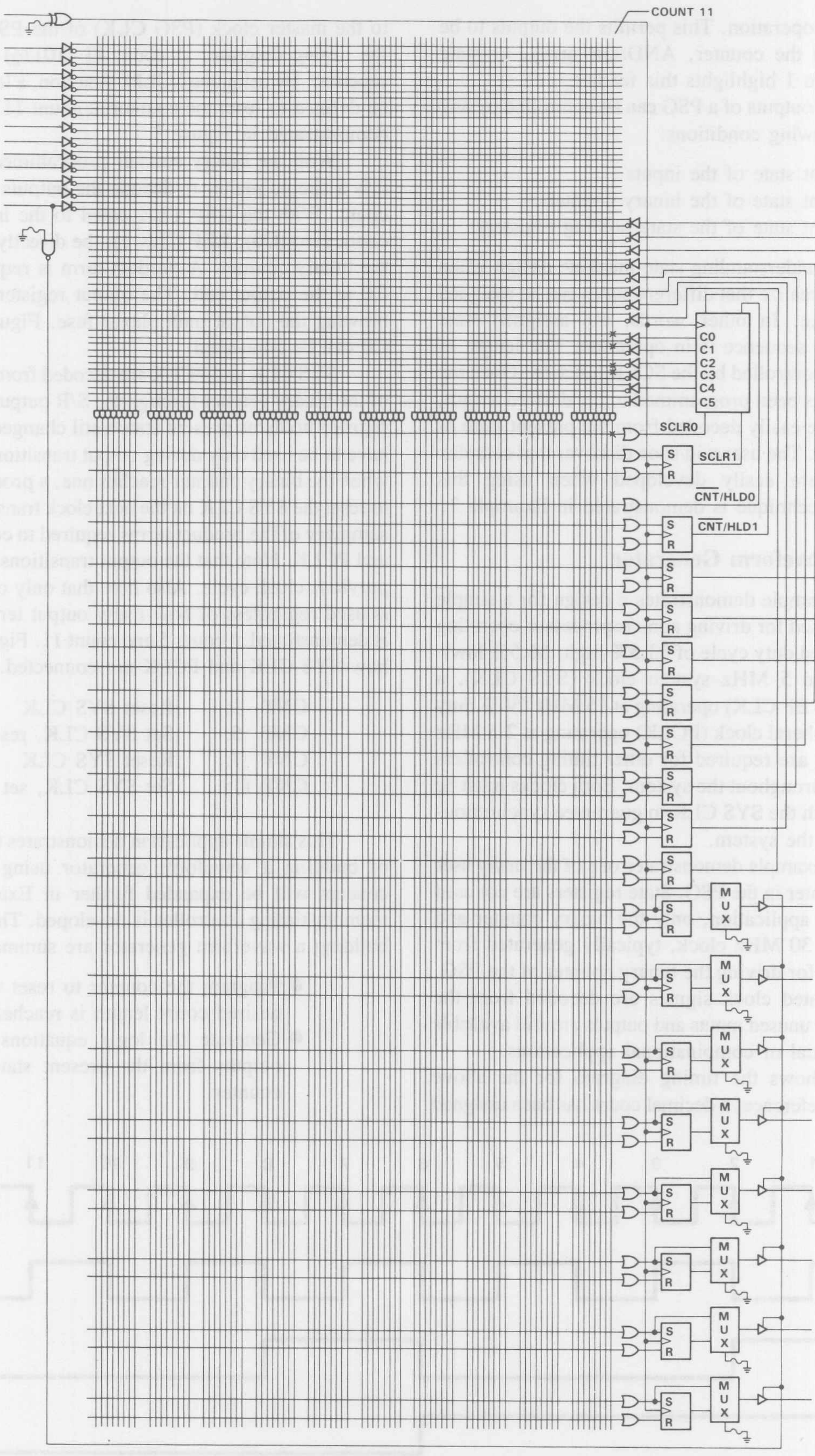


Figure 3. SCLR at COUNT 11
(Example 1 — Waveform Generator)

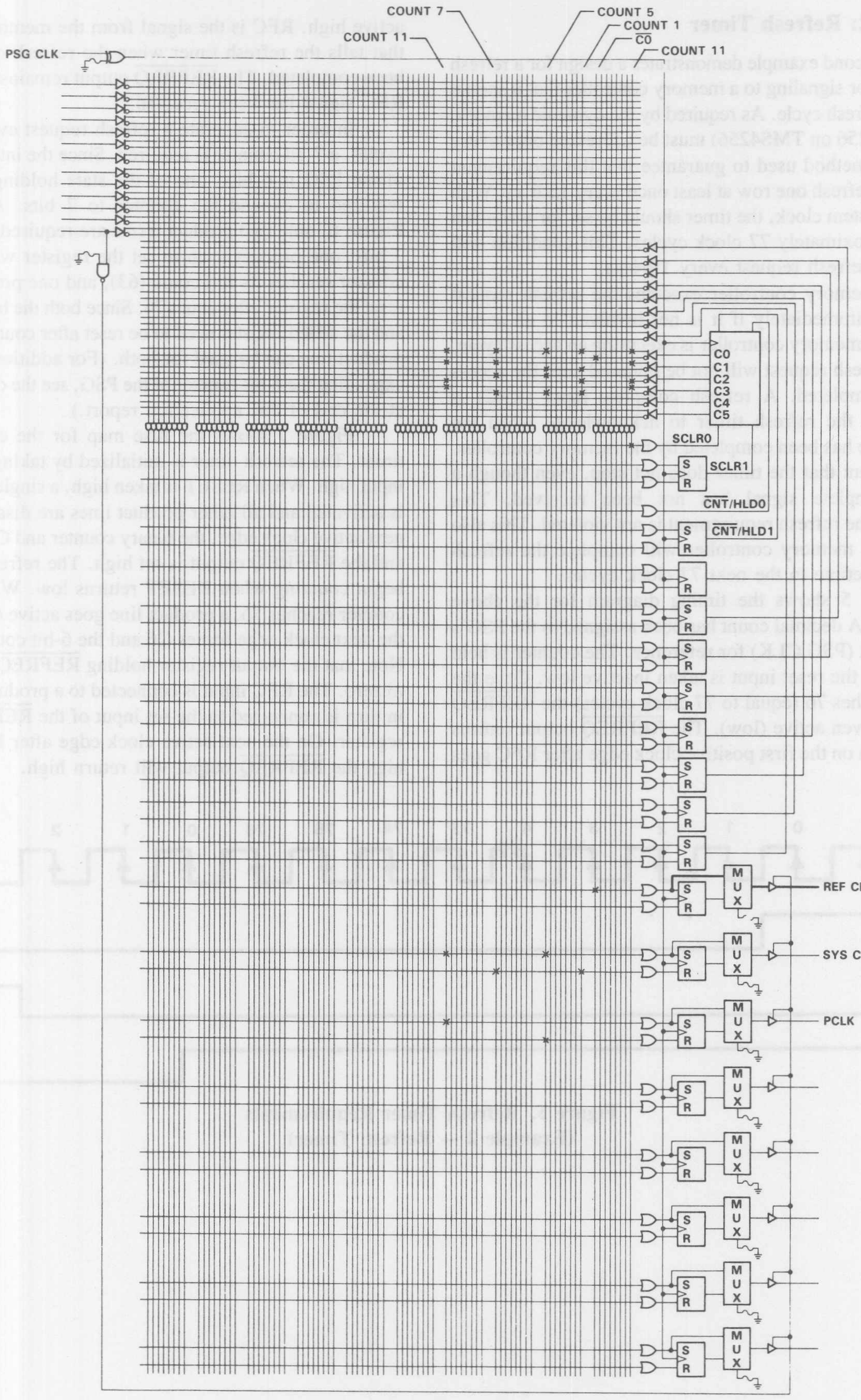


Figure 4. Waveform Generator (Example 1)

Example 2: Refresh Timer

The second example demonstrates a design for a refresh timer used for signaling to a memory controller that it should execute a refresh cycle. As required by the dynamic memory, every row (256 on TMS4256) must be addressed once every 4 ms. One method used to guarantee that this requirement is met is to refresh one row at least once every 15.6 μs. With a 5 MHz system clock, the timer should be set for a division rate of approximately 77 clock cycles. This condition will generate a refresh request every 15.4 μs.

The memory controller executes the refresh request ($\overline{\text{REFREQ}}$) immediately if it is not involved in an access cycle. If the memory controller is executing an access cycle, then the refresh request will not be honored until the access cycle is completed. A refresh complete input (RFC) is required on the refresh timer to acknowledge when the refresh cycle has been completed by the memory controller. It is important that the timer does not stop, even though a refresh complete signal has not been received. This guarantees the refresh requirement is not violated. This also assumes the memory controller will complete the refresh request sometime in the next 77 clock cycles.

Figure 5 shows the timing diagram for the above application. A decimal count has been assigned to the PSG's master clock (PSG CLK) for reference. The counter is held at zero until the reset input is taken inactive low. Once the counter reaches 76 (equal to 77 clock cycles) the $\overline{\text{REFREQ}}$ output is driven active (low). The $\overline{\text{REFREQ}}$ output returns inactive high on the first positive clock edge after RFC goes active high. RFC is the signal from the memory controller that tells the refresh timer when the refresh operation has been completed. The $\overline{\text{REFREQ}}$ output remains low until the RFC signal has been received.

In order to generate a refresh request every 77 clock cycles, a 7-bit counter is required. Since the internal counter of the PSG is 6 bits, one of the state holding registers is required to expand the counter to 7 bits. As shown in Figure 6, only two product terms are required to expand to 7 bits; one product term to set the register when the 6-bit counter reaches its full count (63), and one product term to reset the register after count 76. Since both the binary counter and the added register need to be reset after count 76, a single product line can be used for both. (For additional details on expanding the 6-bit counter of the PSG, see the designer notes at the end of this application report.)

Figure 7 shows the fuse map for the entire refresh timer. The refresh timer is initialized by taking the RESET input high. When RESET is taken high, a single product line is activated and all other product lines are disabled. On the next active clock edge, the binary counter and C6 are cleared and the $\overline{\text{REFREQ}}$ output is set high. The refresh timer will begin counting when RESET returns low. When the 7-bit counter reaches 76, a product line goes active (high) and on the next clock edge forces C6 and the 6-bit counter to zero. Note that the output register holding $\overline{\text{REFREQ}}$ is also reset to zero. The RFC input is connected to a product line which in turn is connected to the set input of the $\overline{\text{REFREQ}}$ output register. On the next active clock edge after RFC is taken high the $\overline{\text{REFREQ}}$ output will return high.

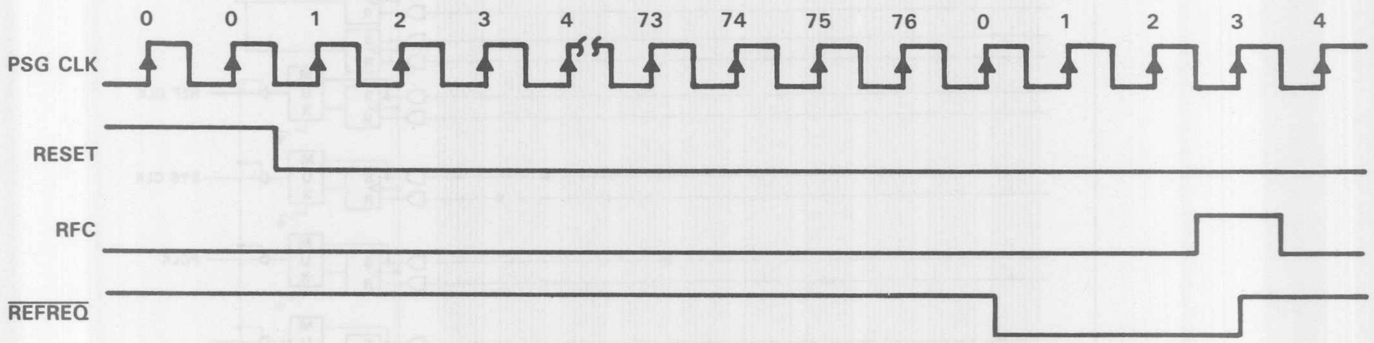


Figure 5. Refresh Timer Requirements (Example 2 — Refresh Timer)

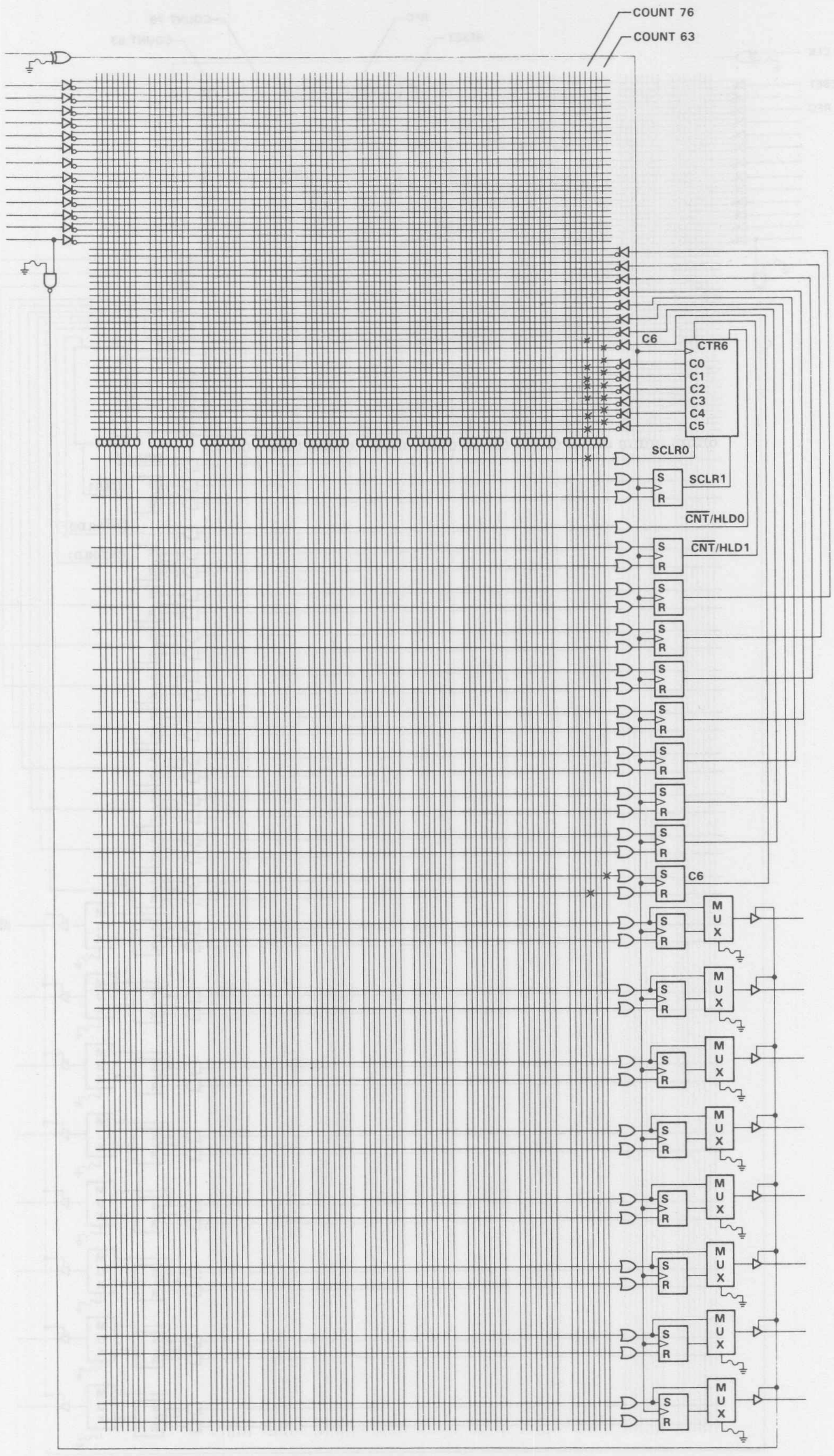


Figure 6. Expanding to 7-Bit Binary Counter
(Example 2 — Refresh Timer)

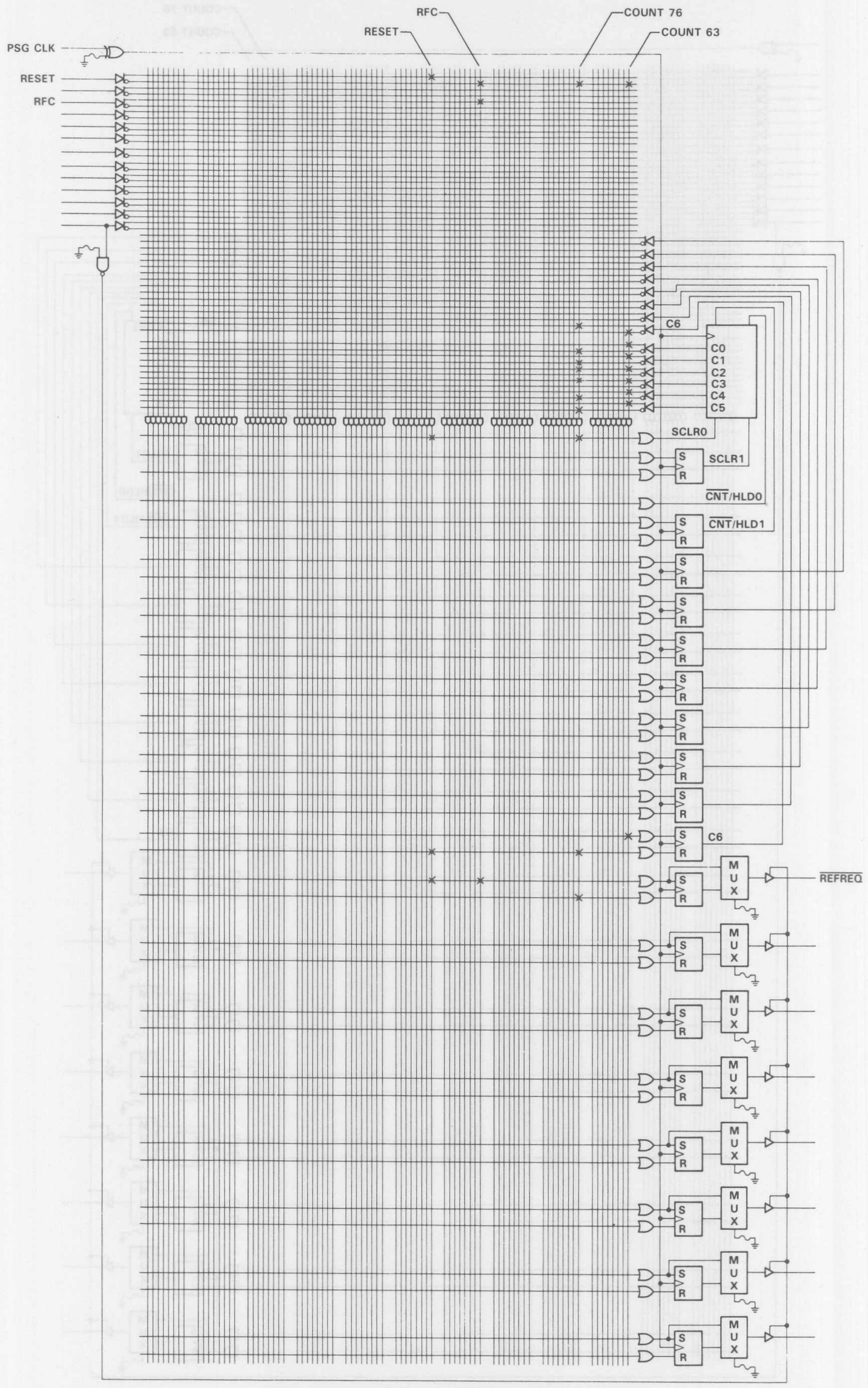

RFC
COUNT 76
RESET
COUNT 63
PSG CLK
RESET
RFC
C6
C0
C1
C2
C3
C4
C5
SCLR0
SCLR1
$\overline{\text{CNT}}$/HLD0
$\overline{\text{CNT}}$/HLD1
S
R
C6
MUX
$\overline{\text{REFREQ}}$

Figure 7. Refresh Timer
(Example 2)

Example 3: Dynamic Memory Timing Controller

The third and last example will demonstrate a state machine design using the PSG507. Figure 8 shows the circuit requirement for a memory timing controller used for interfacing an Intel 8086 to an 'ALS2967 dynamic memory controller. Note that the clock generator and refresh timer, developed in Examples 1 and 2, can be used in this circuit.

The dynamic memory timing controller generates the control signals ($\overline{\text{RAS}}$, $\overline{\text{CAS}}$, MSEL, etc.) needed for accessing and refreshing the dynamic memory. The memory timing controller must also be capable of arbitrating between refresh and access cycles. In other words, if a refresh request ($\overline{\text{REFREQ}}$) occurs while the timing controller is performing an access cycle, the controller must finish the access cycle before granting the refresh request. Likewise, if an access cycle is requested during a refresh cycle, the controller must hold the processor while completing the refresh cycle. After the refresh cycle has been completed, the access cycle can be performed.

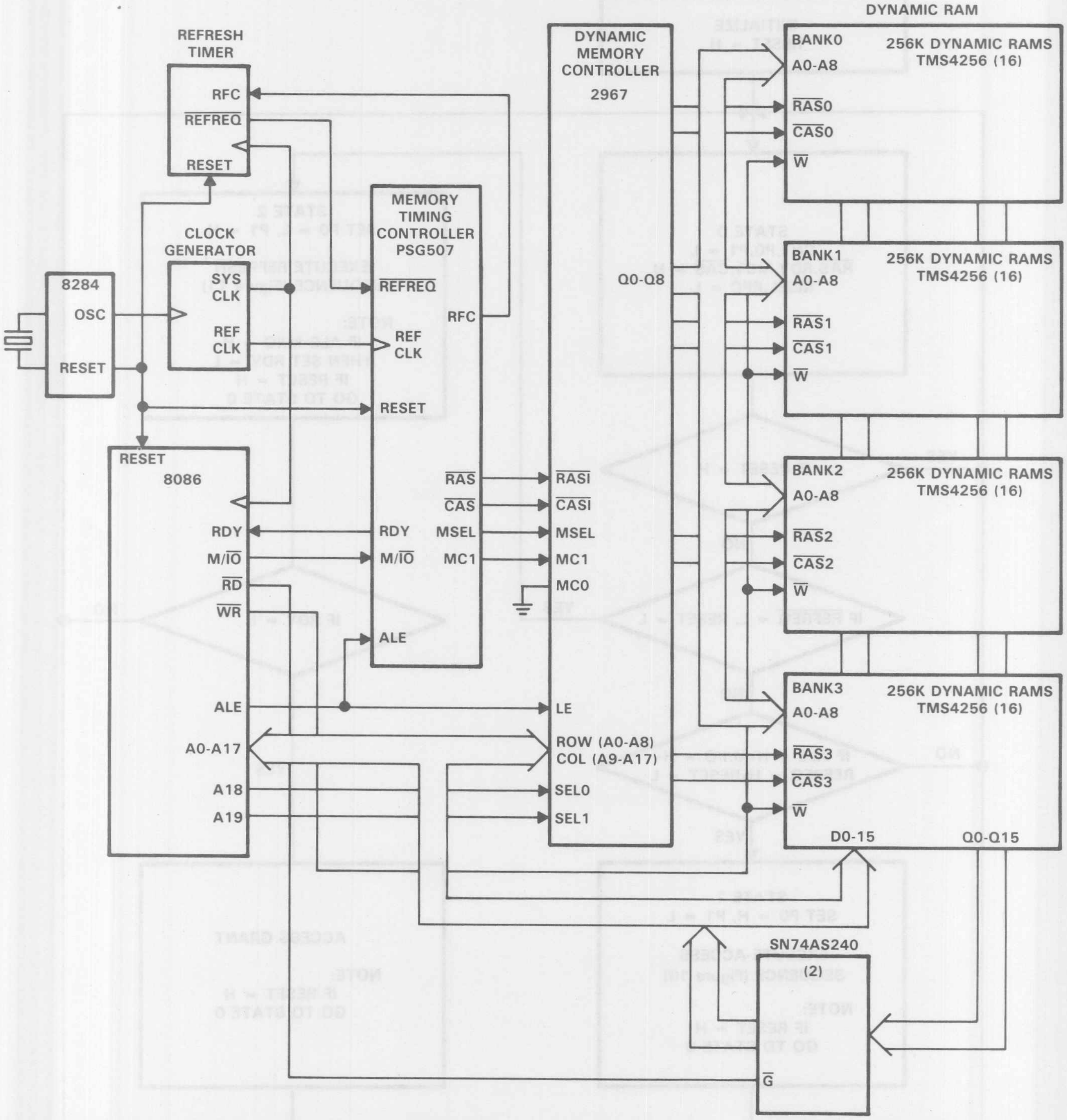


Figure 8. Memory Timing Controller (Example 3)

Figure 9 shows a detailed flow chart for the intended application. Note that two sequences are executed and three states are used. State 0 (ST0) provides an initalization and holding state, while state 1 (ST1) is assigned to the access sequence. The access sequence consists of 10 clock cycles as shown in Figure 10. State 2 (ST2) is assigned to the refresh/access grant sequence (Figure 11). This particular sequence takes 20 clock cycles, with a logical decision being made between count 9 and count 10. If at count 9 RDY is low, the counter will continue on and execute the access grant sequence. If RDY is high, the controller will clear the counter and return to state 0.

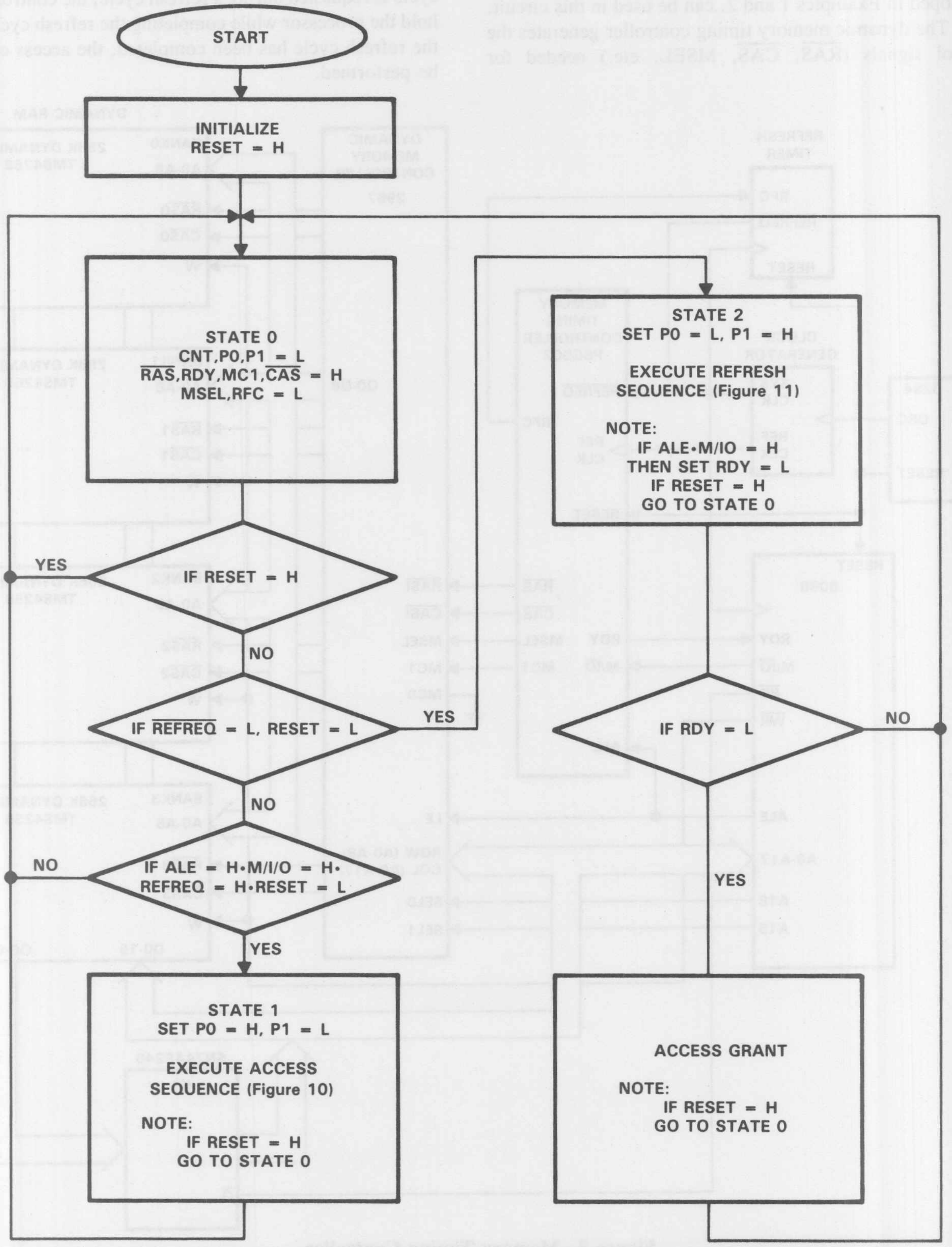


Figure 9. Flow Chart: Dynamic Memory Timing Controller

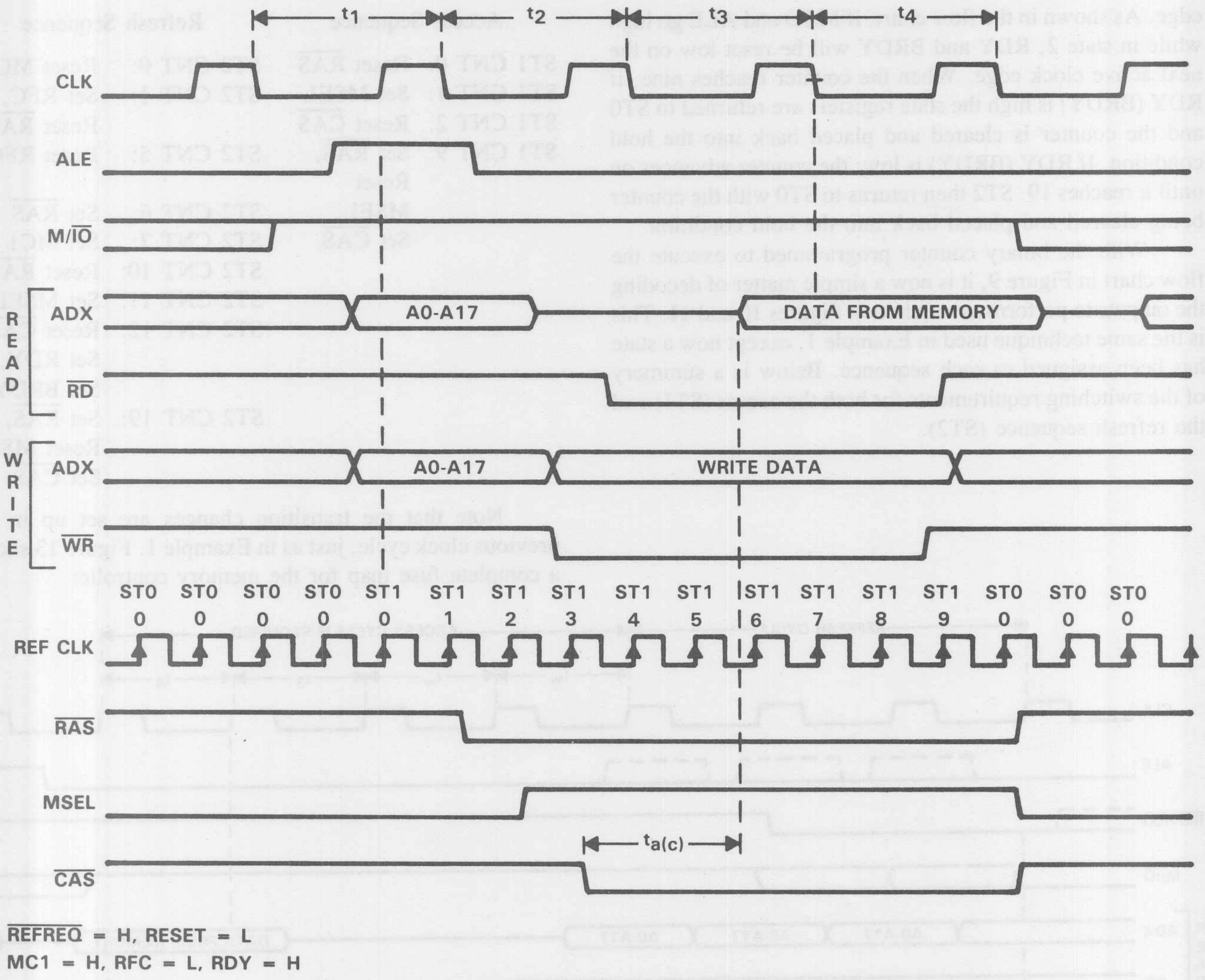


Figure 10. Access Cycle

Developing the logic equations for this application becomes a simple matter when referencing the sequences to a decimal count (Figures 10 and 11). It is important to realize that each sequence has been referenced to a state. This allows the same binary counter to be used for each sequence, even though each sequence is of a different length.

The first step in implementing the above application is to define the logic equations which will make the binary counter perform as described in the flow chart of Figure 9. As will become evident, these equations fall directly from the flow chart. After the counter has been made to perform as described, the outputs can be easily decoded from the binary count and the present state of the state holding registers.

Figure 12 shows a fuse map for step 1 as described above. Initalization is performed by taking the reset input high. When this condition occurs, all product lines except the reset product line are forced inactive. When the reset product line is active, the counter and state holding registers (P0 and P1) are reset to zero on the first active clock edge. The $\overline{CNT}$/HLD1 register is set high, which places the counter in the hold mode. The RDY, MC1, $\overline{RAS}$, and $\overline{CAS}$ outputs are driven high on the same active clock edge. Since the RDY output does not feed back to the AND array, a buried state register, BRDY, is used to monitor the RDY output and is also set high. MSEL and RFC are driven low.

Controlling the binary counter is a simple matter and normally takes only a couple of logic equations. For each sequence, a start and stop condition must be defined. In the case of ST1, when the condition RESET = L, ALE = H, M/$\overline{IO}$ = H, $\overline{REFREQ}$ = H, P0 = L, and P1 = L occurs, ST0 (P1 = L, P0 = L) changes to ST1 (P1 = L, P0 = H), and the $\overline{CNT}$/HLD1 register is driven low to let the counter advance on the next active clock edge. When the counter reaches nine, ST1 returns to ST0 and the counter is cleared and put back into the hold condition.

In the case of ST2, when the condition RESET = L, $\overline{REFREQ}$ = L, P0 = L, and P1 = L occurs, ST0 changes to ST2 (P1 = H, P0 = L) and the $\overline{CNT}$/HLD1 register is driven low to let the counter advance on the next active clock edge. As shown in the flow chart, if M/IO and ALE go high while in state 2, RDY and BRDY will be reset low on the next active clock edge. When the counter reaches nine, if RDY (BRDY) is high the state registers are returned to ST0 and the counter is cleared and placed back into the hold condition. If RDY (BRDY) is low, the counter advances on until it reaches 19. ST2 then returns to ST0 with the counter being cleared and placed back into the hold condition.

With the binary counter programmed to execute the flow chart in Figure 9, it is now a simple matter of decoding the outputs to perform as required in Figures 10 and 11. This is the same technique used in Example 1, except now a state has been assigned to each sequence. Below is a summary of the switching requirements for both the access (ST1) and the refresh sequence (ST2).

Access Sequence		Refresh Sequence	
ST1 CNT 0:	Reset $\overline{RAS}$	ST2 CNT 0:	Reset MC1
ST1 CNT 1:	Set MSEL	ST2 CNT 1:	Set RFC, Reset $\overline{RAS}$
ST1 CNT 2:	Reset $\overline{CAS}$		
ST1 CNT 9:	Set $\overline{RAS}$, Reset MSEL, Set $\overline{CAS}$	ST2 CNT 5:	Reset RFC
		ST2 CNT 6:	Set $\overline{RAS}$
		ST2 CNT 7:	Set MC1
		ST2 CNT 10:	Reset $\overline{RAS}$
		ST2 CNT 11:	Set MSEL
		ST2 CNT 12:	Reset $\overline{CAS}$, Set RDY, Set BRDY
		ST2 CNT 19:	Set $\overline{RAS}$, Reset MSEL, Set $\overline{CAS}$

Note that the transition changes are set up in the previous clock cycle, just as in Example 1. Figure 13 shows a complete fuse map for the memory controller.

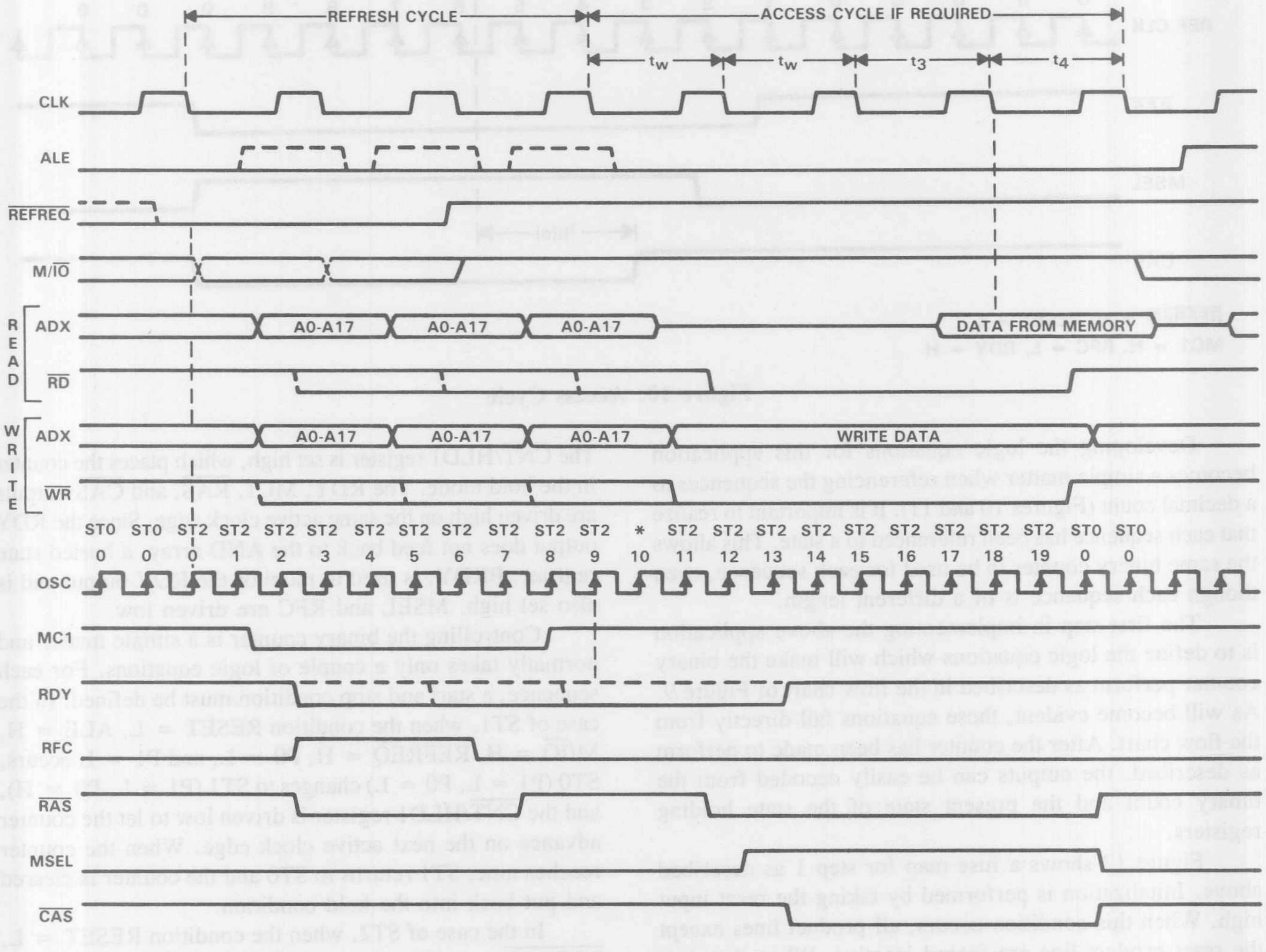


Figure 11. Refresh/Access Grant Cycle

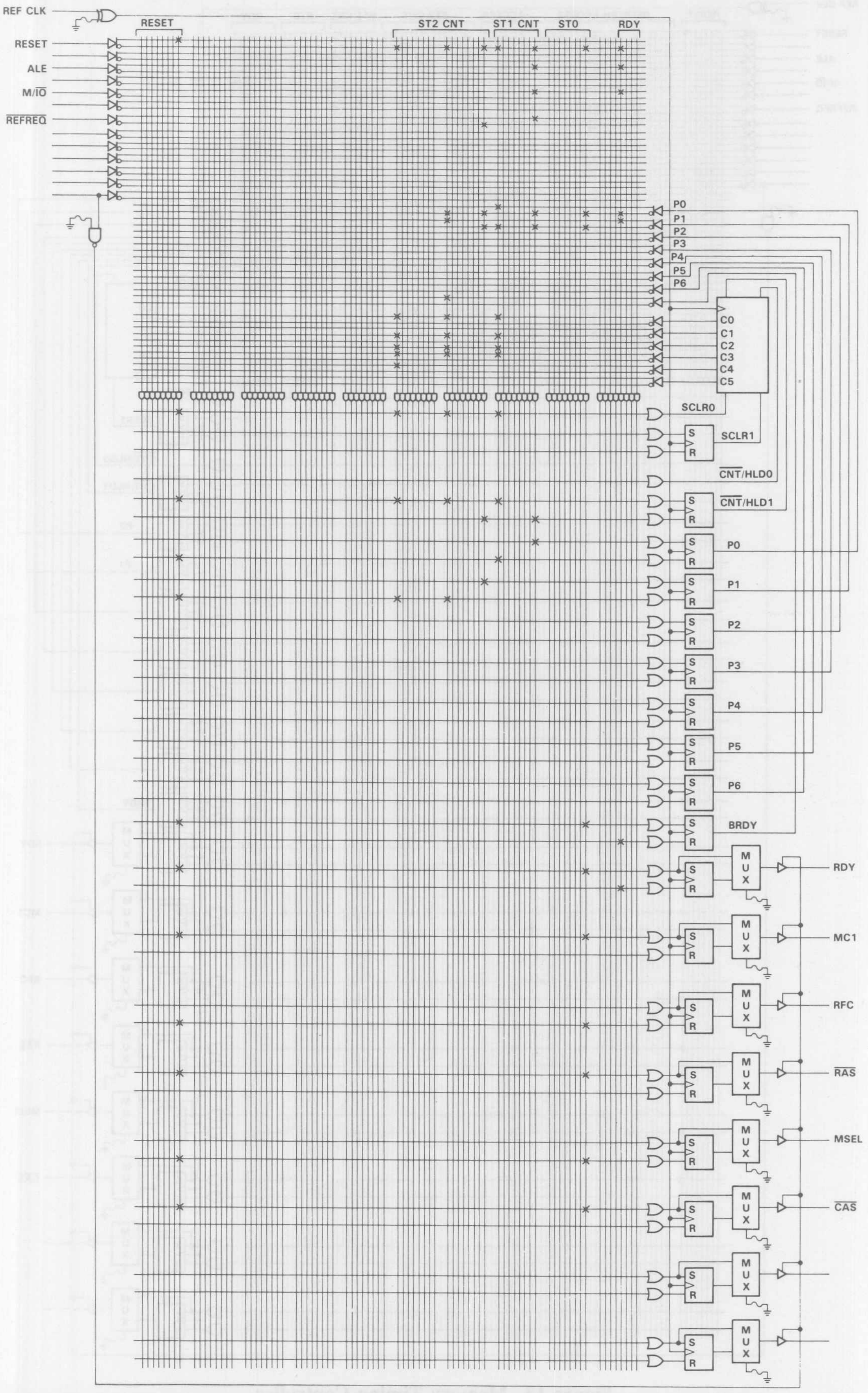


Figure 12. Counter Control Logic (Example 3 — Dynamic Memory Timing Controller)

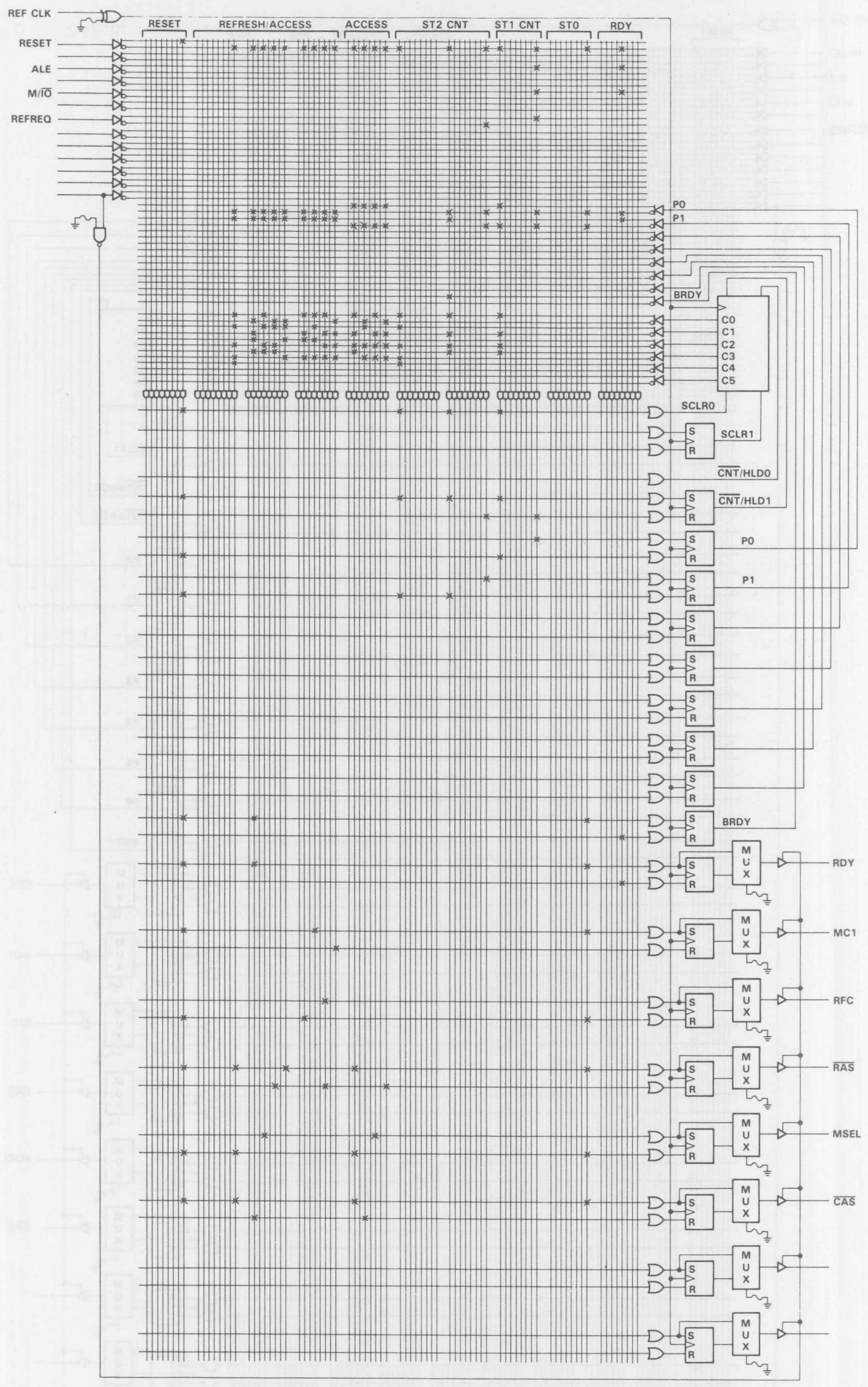


Figure 13. Memory Timing Controller (Example 3)

DESIGNER NOTES

Obtaining Maximum Counter Performance

As with any programmable logic device, there are usually several different methods for implementing any one application. In some cases, device performance is affected. On the PSG, maximum counter frequency is affected by how the designer controls the 6-bit counter.

For example, in the waveform generator example shown at the beginning of this application note, the counter was reset to zero after reaching count 11 by using the nonregistered SCLR0 function. By using the registered SCLR1 function, a higher operating frequency is obtainable.

This method requires an additional "AND" term as shown in Figure 14, but does provide maximum performance. Note that during the 10th clock cycle the set input on the SCLR1 register is high. On the next active clock edge, the counter advances to 11 and the SCLR1 register is set high. This causes the counter to be reset on the next active clock edge. At the same time, the SCLR1 register is reset low to allow the counter to advance past zero.

In effect, the setup time requirement for SCLR1 is performed in the previous clock cycle. When using the SCLR0 method, the setup time must be added to the f_{max} equation. This results in a lower f_{max}. The same tradeoffs apply with the $\overline{CNT}$/HLD function. The PSG507 data sheet specifies f_{max} for both methods.

Expanding the 6-Bit Counter

In Example 2, the six bit counter had to be expanded to 7 bits. This was accomplished by adding one of the state registers to the most significant bit of the counter. It should be noted that the synchronous clear and count hold functions must be controlled through the set and reset inputs of the added bits. The designer must be aware of certain limitations when trying to perform this function. Figure 15 shows three additional bits being added to the 6-bit counter. Note that every bit added requires two additional "AND" terms.

A problem can arise on certain counts when trying to generate a synchronous clear before reaching the full binary count (all outputs high). The designer must ensure that both S and R are not high simultaneously. For example, let's say we want the 9-bit counter to return to zero at count 383 (101111111_2). At count 383, the S/R register used for C7 is being told to set. Therefore, any reset command would result in both S and R being used simultaneously.

This problem, only seen on a few data words, can be solved by using another state register to control the counter reset. This method is similar to that used above to obtain maximum operating frequency. Figure 16 shows the 9-bit counter returning to zero after count 383. Notice that at count 382 the extra S/R register is being told to reset on the next active clock edge. At count 383 the six product lines controlling C6, C7, and C8 are disabled by the feedback from the extra register, in particular the S input on C7. At count 383, the 9-bit counter will return to zero and the extra register is set high.

An extra register may also be needed to achieve the count/hold function when using an expanded counter. During certain counts the added bits will change state, even though the 6-bit counter is programmed to hold. For example, let's say we want the 9-bit counter to hold at count 383. Even though the 6-bit counter can be held at 111111, C6 and C7 will advance on the next active clock edge. In order to hold C6 and C7 where they are, an extra register is used to disable the product lines responsible for the transition from count 383 to 384. Since the counter is on hold, the extra hold register can only be reset from an input pin or a state register(s) transition (not on the next count). In this example, an input pin is used to reset the extra register and the $\overline{CNT}$/HOLD register. When the CONTINUE input is taken low, the counter will continue to advance. The system must guarantee that the continue input will not be low during count 382 to avoid the indeterminant set = H, reset = H state. Figure 17 shows this 9-bit counter.

It is also important to note that when using extra registers a reset input may be necessary to set the extra registers high after powerup, since all S/R registers power-up clear. This requirement would not be necessary if the phase of the extra register was reversed. This is easily accomplished by using the inverted feedback from the extra register. However, it is good state machine design practice to include a reset input that forces all S/R registers to a known state.

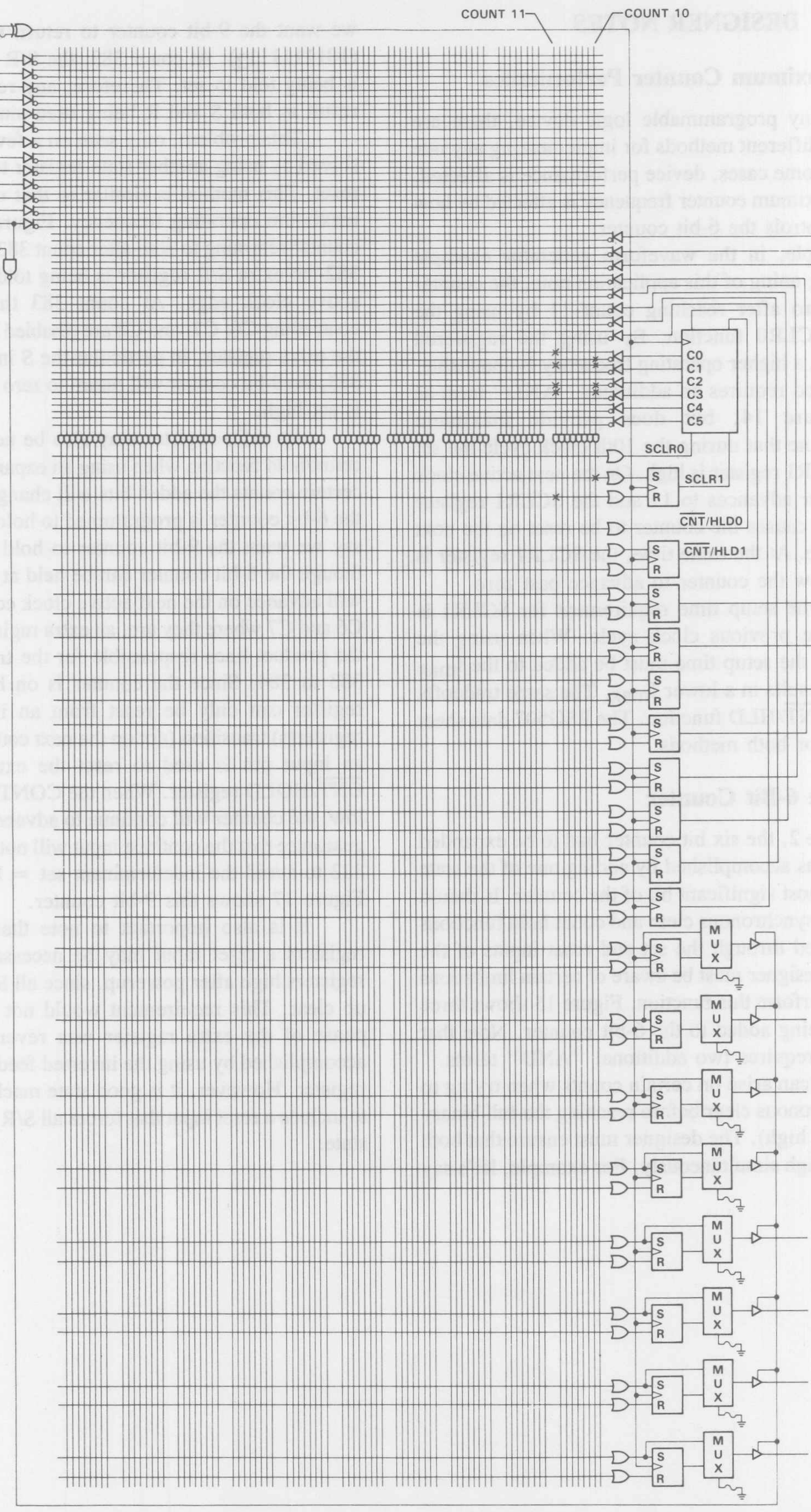


Figure 14. Registered $\overline{\text{SCLR}}$ Example (Designer Notes)

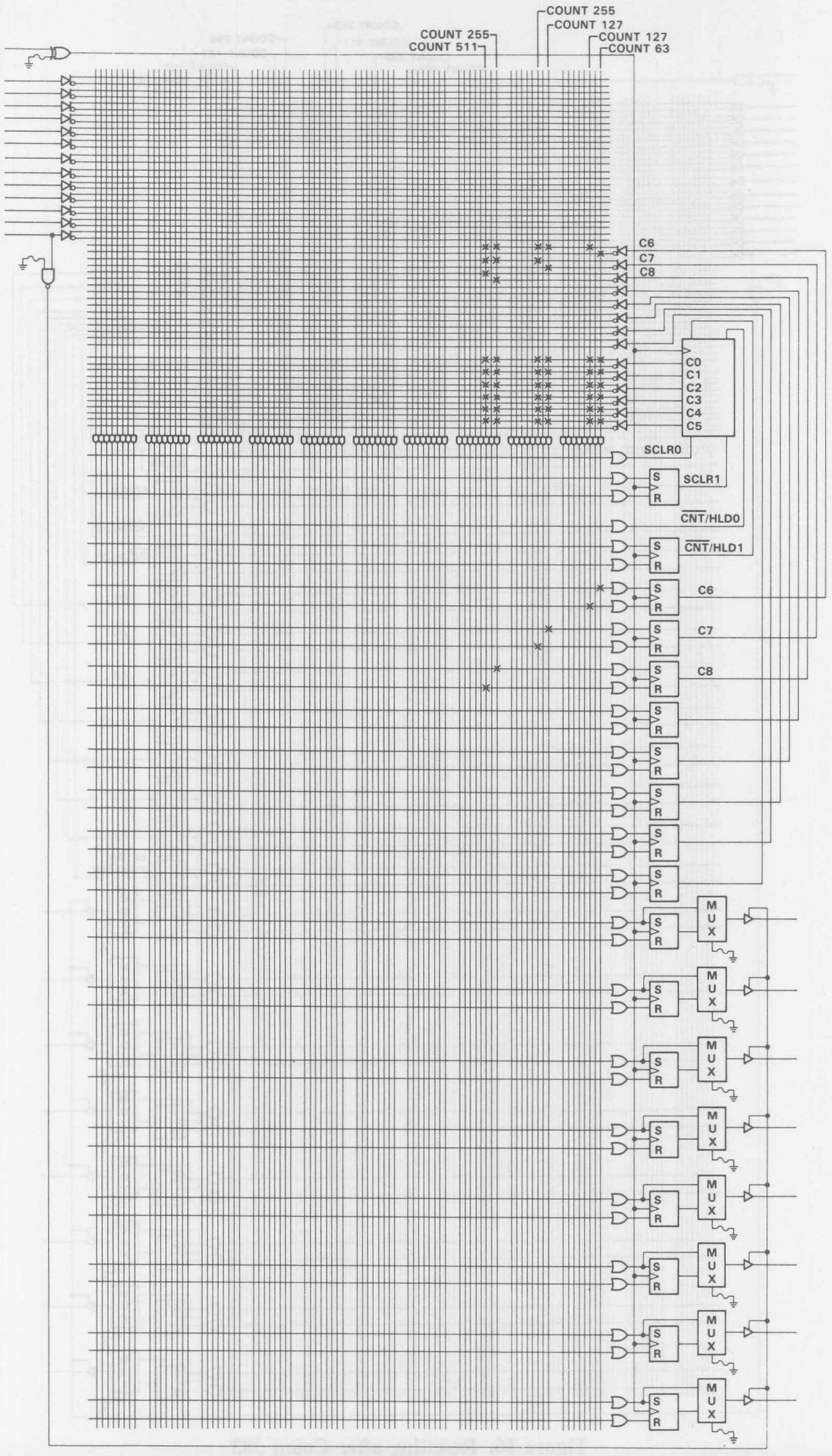


Figure 15. Expanding to 9-Bit Counter

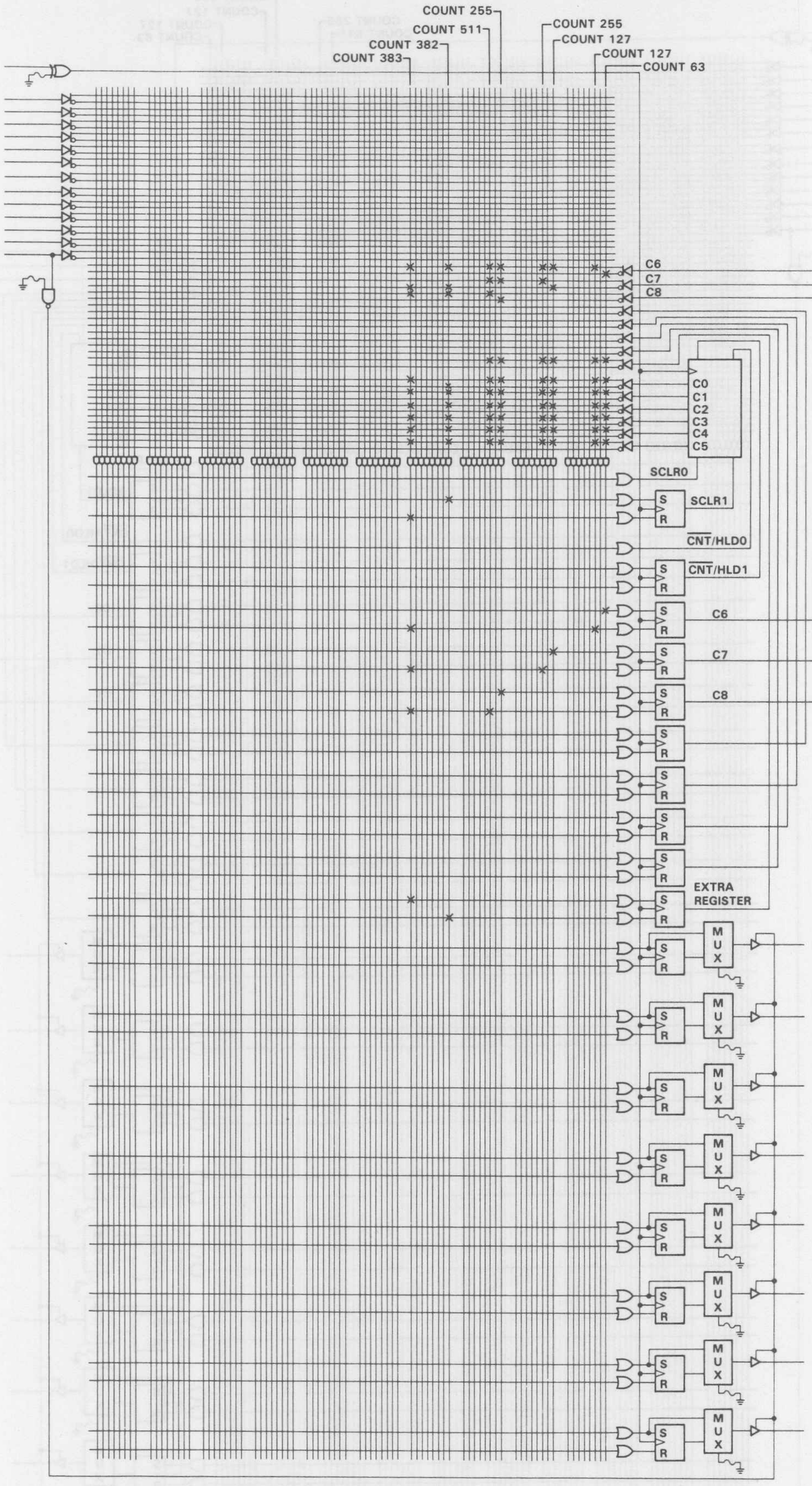


Figure 16. Resetting after Count 383
(Expanding the 6-Bit Counter)

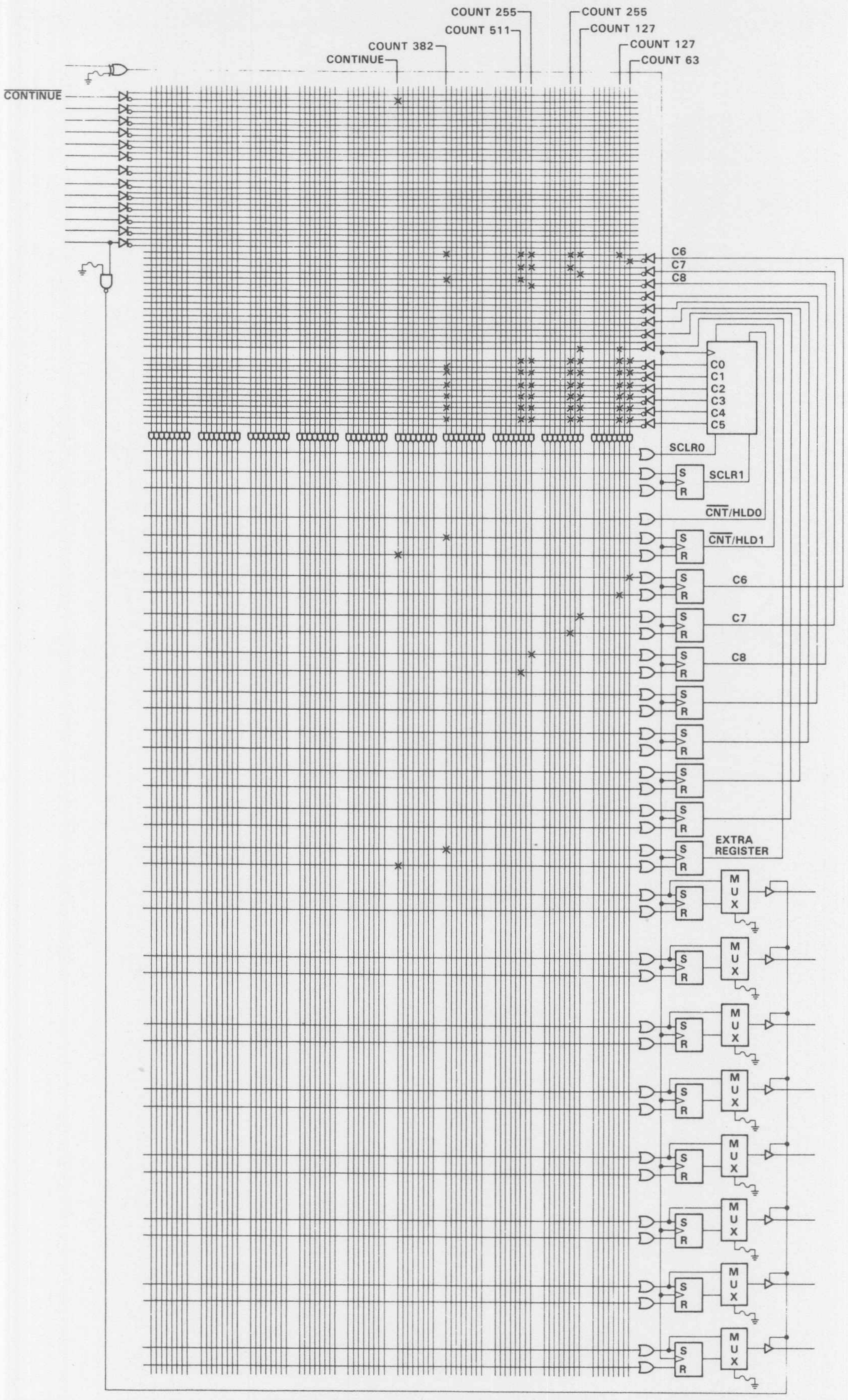


Figure 17. Holding the 9-Bit Counter at Count 383
(Expanding the 6-Bit Counter)

’PSG507 Example 1: Waveform Generator

```
title {

     Device:         TIBPSG507

     Application:    PSG507 Example 1:  Waveform Generator.

     Source:         Breuninger, R. K. "A Designer's Guide to the
                     TIBPSG507," Programmable Logic Data Book, 3-83,
                     Texas Instruments, 1990.

     Transcription: proLogic Systems Inc.,
                     5570 Burbank Street
                     Broomfield, CO 80020
                     303-460-0103
}

include a507;

/* counter states of interest */
define COUNT1 =pt00;
               pt00 = !c3 & !c2 & !c1 &  c0;
define COUNT5 =pt01;
               pt01 = !c3 &  c2 & !c1 &  c0;
define COUNT7 =pt02;
               pt02 = !c3 &  c2 &  c1 &  c0;
define COUNT11=pt03;
               pt03 =  c3 & !c2 &  c1 &  c0;

/* ref_clk */  pin8    = pt04;
                         pt04 = !c0;

/* sys_clk */  pin9.s = COUNT5 | COUNT11;
               pin9.r = COUNT1 | COUNT7;

/* pclk */     pin10.s= COUNT11;
               pin10.r= COUNT5;

   sclr0               = COUNT11;
                                  /* synchronous clear on count 11 */

test_vectors { /* psg_clk    count      ref_clk  sys_clk  pclk  */

                  pin1                  pin8      pin9   pin10;

                    0      /*  0 */      H         L       L  ;
                    C      /*  1 */      L         L       L  ;
                    C      /*  2 */      H         L       L  ;
                    C      /*  3 */      L         L       L  ;
                    C      /*  4 */      H         L       L  ;
                    C      /*  5 */      L         L       L  ;
                    C      /*  6 */      H         H       L  ;
```

```
          C     /*  7 */     L         H      L ;
          C     /*  8 */     H         L      L ;
          C     /*  9 */     L         L      L ;
          C     /* 10 */     H         L      L ;
          C     /* 11 */     L         L      L ;
          C     /*  0 */     H         H      H ;
          C     /*  1 */     L         H      H ;
          C     /*  2 */     H         L      H ;
          C     /*  3 */     L         L      H ;
          C     /*  4 */     H         L      H ;
          C     /*  5 */     L         L      H ;
          C     /*  6 */     H         H      L ;
          C     /*  7 */     L         H      L ;
          C     /*  8 */     H         L      L ;
          C     /*  9 */     L         L      L ;
          C     /* 10 */     H         L      L ;
          C     /* 11 */     L         L      L ;

          C     /*  0 */     H         H      H ;
}
```

’PSG507 Example 2: Refresh Timer

```
title {

     Device:         TIBPSG507

     Application:    PSG507 Example 2:  Refresh Timer.

     Source:         Breuninger, R. K. "A Designer's Guide to the
                     TIBPSG507," Programmable Logic Data Book, 3-86,
                     Texas Instruments, 1990.

     Transcription: proLogic Systems Inc.,
                     5570 Burbank Street
                     Broomfield, CO 80020
                     303-460-0103
}

include a507;

/* input signals */
define  reset   = pin2;  /* inactive low */
define  RESET   =        pt00;
                         pt00 = reset;

define  rfc     = pin3;  /* active high refresh complete */
define  RFC     =        pt01;
                         pt01 = rfc;

/* the seventh counter bit */
define  c6      = p0;

/* counter states of interest */
define COUNT63  = pt02;
     pt02 = !c6.q &  c5 &  c4 &  c3 &  c2 &  c1 &  c0 & !reset;
define COUNT76  = pt03;
     pt03 =  c6.q & !c5 & !c4 &  c3 &  c2 & !c1 & !c0 & !reset;

/* counter control */
   c6.s         = COUNT63;
   c6.r         = COUNT76 | RESET;
   sclr0        = COUNT76 | RESET;

/* refreq */
define refreq = pin8;    refreq.s = RFC | RESET;
                         refreq.r = COUNT76;

test_vectors {

                  pin1  reset  rfc  refreq;

                   0      1     0      L;     /* power on */
  repeat  4 {      C      1     0      H;   } /* reset */
repeat 76 {     C       0       0       H;      }
                C       0       0       L;
                C       0       0       L;
                C       0       1       H;
repeat 74 {     C       0       0       H;      }
                C       0       0       L;
                C       0       0       L;
                C       0       1       H;
}
```

'PSG507 Example 3: Dynamic Memory Timing Controller

```
title {

    Device:         TIBPSG507

    Application:    PSG507 Example 3:  Dynamic Memory Timing
                    Controller.

    Source:         Breuninger, R. K. "A Designer's Guide to the
                    TIBPSG507," Programmable Logic Data Book, 3-89,
                    Texas Instruments, 1990.

    Transcription: proLogic Systems Inc.,
                    5570 Burbank Street
                    Broomfield, CO 80020
                    303-460-0103
}

include a507;

/* input signals */
define  reset   = pin2;  /* inactive low */
define  ale     = pin3;  /* address latch enable */
define  mio     = pin4;  /* memory I/O */
define  refreq  = pin5;  /* refresh request */

/* output signals */
define  rdy     = pin8;  /* ready */
define  mc1     = pin9;  /* mode control */
define  rfc     = pin10; /* refresh complete */
define  ras     = pin11; /* row address strobe */
define  msel    = pin13; /* multiplexer select */
define  cas     = pin14; /* column address strobe */

/* internal */
define  brdy    = p2;    /* buried ready - always identical to
                            output signal 'rdy'.  Permits testing
                            the output pin state. */

/* counter states of interest */
define COUNT0  = !c4 & !c3 & !c2 & !c1 & !c0;
define COUNT1  = !c4 & !c3 & !c2 & !c1 &  c0;
define COUNT2  = !c4 & !c3 & !c2 &  c1 & !c0;
define COUNT3  = !c4 & !c3 & !c2 &  c1 &  c0;
define COUNT5  = !c4 & !c3 &  c2 & !c1 &  c0;
define COUNT6  = !c4 & !c3 &  c2 &  c1 & !c0;
define COUNT9  = !c4 &  c3 & !c2 & !c1 &  c0;
define COUNT10 = !c4 &  c3 & !c2 &  c1 & !c0;
define COUNT11 = !c4 &  c3 & !c2 &  c1 &  c0;
define COUNT12 = !c4 &  c3 &  c2 & !c1 & !c0;
define COUNT19 =  c4 & !c3 & !c2 &  c1 &  c0;
```

```
/* LOW and HIGH operations for clarity */
define LOW  = .r=1; /* usage: (rs LOW) -> (rs.r=1) */
define HIGH = .s=1;

state_diagram (p1.q,p0.q) {

    if (reset) {
        /* These are the levels of all output and control
      signals in the idle state.  Other states return
          modified signals to these levels before resuming the
          idle state. */

        mc1 HIGH; rdy HIGH; rfc  LOW; ras HIGH; msel LOW;
                  cas HIGH; brdy HIGH;
        sclr0=1;  hld1 HIGH; /* counter cleared and holding */
        idleState;
    }

    state idleState=00 {
        /* Wait for Request */
        if (ale & mio & refreq) {
            /* Memory Access Request */
            hld1 LOW;
            accessCycle;
        }
        if (!refreq) {
            /* Memory Refresh Request */
            hld1 LOW;
            refreshCycle;
        }
    }

    state accessCycle=01 {
        /* Generate the Memory Access Sequence. */
        if (COUNT0)
            ras LOW;
        if (COUNT1)
            msel HIGH;
        if (COUNT2)
            cas LOW;
        if (COUNT9) {
            /* Return modified control and output
              signals to their idle values. */
            ras HIGH; msel LOW; cas HIGH;
            sclr0=1;  hld1 HIGH;
            idleState;
        }
    }
    state refreshCycle=1x {
        /* Generate the Memory Refresh Sequence. */
        if (ale & mio) {
            /* An Access Request occurs during refresh.  Hold
               off the processor until refresh is complete (at
               COUNT9 below). */
            rdy  LOW;
            brdy LOW;
        }
        if (COUNT0)
            mc1 LOW;
        if (COUNT1) {
            rfc HIGH; ras LOW;
        }
        if (COUNT3)
            mc1 HIGH;
        if (COUNT5)
            rfc LOW;
        if (COUNT6)
            ras HIGH;
        if (COUNT9)
            /* The Refresh Sequence is complete. */
            if (brdy.q) {
                /* The processor did NOT make a Memory Access
                   request during the Refresh Sequence.
                   Return to idle. */
                sclr0=1; hld1 HIGH;
                idleState;
            }
        /* A Memory Access Request was received during the
           Refresh Sequence.  Generate the Memory Access
           Sequence now. */
        if (COUNT10)
            ras LOW;
        if (COUNT11)
            msel HIGH;
        if (COUNT12) {
            rdy  HIGH; cas LOW;
            brdy HIGH;
        }
        if (COUNT19) {
            ras HIGH; msel LOW; cas HIGH;
            sclr0=1;  hld1 HIGH;
            idleState;
        }
    }
}

test_vectors {

            /* Access Cycle */
```

```
pin1 reset ale mio refreq rdy mc1 rfc ras msel cas;
 0     0    0   0    1     L   L   L   L   L    L ; /* power on*/

 C     1    0   0    1     H   H   L   H   L    H ; /* reset */
 C     0    0   0    1     H   H   L   H   L    H ;

 C     0    0   0    1     H   H   L   H   L    H ;
 C     0    0   1    1     H   H   L   H   L    H ;
 C     0    1   1    1     H   H   L   H   L    H ; /*  0 */
 C     0    0   1    1     H   H   L   L   L    H ; /*  1 */
 C     0    0   1    1     H   H   L   L   H    H ; /*  2 */
 C     0    0   1    1     H   H   L   L   H    L ; /*  3 */
 C     0    0   1    1     H   H   L   L   H    L ; /*  4 */
 C     0    0   1    1     H   H   L   L   H    L ; /*  5 */
 C     0    0   1    1     H   H   L   L   H    L ; /*  6 */
 C     0    0   1    1     H   H   L   L   H    L ; /*  7 */
 C     0    0   1    1     H   H   L   L   H    L ; /*  8 */
 C     0    0   1    1     H   H   L   L   H    L ; /*  *0 */
 C     0    0   1    1     H   H   L   H   L    H ; /*  0 */
 C     0    0   1    1     H   H   L   H   L    H ; /*  0 */
               /* Refresh Cycle

pin1 reset ale mio refreq rdy mc1 rfc ras msel cas        */

 C     0    0   0    0     H   H   L   H   L    H ; /*  0 */
 C     0    0   1    0     H   L   L   H   L    H ; /*  1 */
 C     0    0   1    0     H   L   H   L   L    H ; /*  2 */
 C     0    0   1    0     H   L   H   L   L    H ; /*  3 */
 C     0    0   1    0     H   H   H   L   L    H ; /*  4 */
 C     0    0   1    0     H   H   H   L   L    H ; /*  5 */
 C     0    0   1    1     H   H   L   L   L    H ; /*  6 */
 C     0    0   1    1     H   H   L   H   L    H ; /*  7 */
 C     0    0   1    1     H   H   L   H   L    H ; /*  8 */
 C     0    0   1    1     H   H   L   H   L    H ; /*  9 */
 C     0    0   1    1     H   H   L   H   L    H ; /*  0 */
 C     0    0   1    1     H   H   L   H   L    H ; /*  0 */

               /* Refresh/Access Grant Cycle

pin1 reset ale mio refreq rdy mc1 rfc ras msel cas        */

 C     0    0   0    0     H   H   L   H   L    H ; /*  0 */
 C     0    0   1    0     H   L   L   H   L    H ; /*  1 */
 C     0    0   1    0     H   L   H   L   L    H ; /*  2 */
 C     0    1   1    0     L   L   H   L   L    H ; /*  3 */
 C     0    0   1    0     L   H   H   L   L    H ; /*  4 */
 C     0    0   1    0     L   H   H   L   L    H ; /*  5 */
 C     0    0   1    1     L   H   L   L   L    H ; /*  6 */
 C     0    0   1    1     L   H   L   H   L    H ; /*  7 */
 C     0    0   1    1     L   H   L   H   L    H ; /*  8 */
 C      0    0   1    1     L   H   L   H   L    H ; /*  9 */
 C      0    0   1    1     L   H   L   H   L    H ; /* 10 */
 C      0    0   1    1     L   H   L   L   L    H ; /* 11 */
 C      0    0   1    1     L   H   L   L   H    H ; /* 12 */
 C      0    0   1    1     H   H   L   L   H    L ; /* 13 */
 C      0    0   1    1     H   H   L   L   H    L ; /* 14 */
 C      0    0   1    1     H   H   L   L   H    L ; /* 15 */
 C      0    0   1    1     H   H   L   L   H    L ; /* 16 */
 C      0    0   1    1     H   H   L   L   H    L ; /* 17 */
 C      0    0   1    1     H   H   L   L   H    L ; /* 18 */
 C      0    0   1    1     H   H   L   L   H    L ; /* 19 */
 C      0    0   1    1     H   H   L   H   L    H ; /*  0 */
 C      0    0   1    1     H   H   L   H   L    H ; /*  0 */
}
```

Appendix D

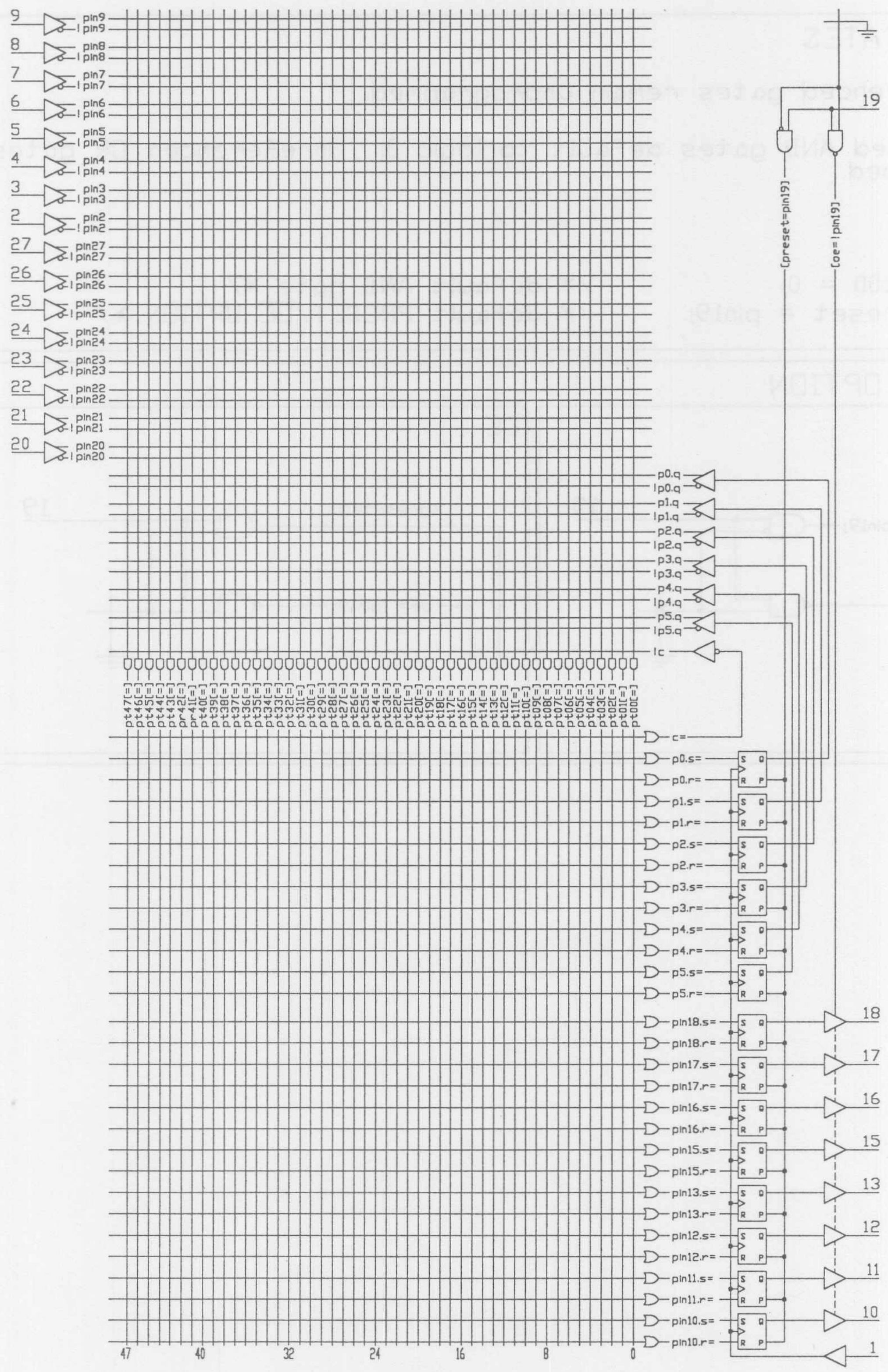
TI Device Cross Reference

TI DEVICE CROSS REFERENCE

TI Device Name	proLogic Name	Page
EP330	P330	E-30
EP630	P630	E-32
EP1830	P1830	E-35
PAL16L8A/A–2	P16L8	E-15
PAL16R4A/A–2	P16R4	E-17
PAL16R6A/A–2	P16R6	E-18
PAL16R8A/A–2	P16R8	E-19
PAL20L8A	P20L8	E-22
PAL20R4A	P20R4	E-23
PAL20R6A	P20R6	E-24
PAL20R8A	P20R8	E-25
TIBPAD16N8–7	P16N8	E-16
TIBPAD18N8–6	P18N8	E-20
TIBPAL16L8	P16L8	E-15
TIBPAL16R4	P16R4	E-17
TIBPAL16R6	P16R6	E-18
TIBPAL16R8	P16R8	E-19
TIBPAL20L8	P20L8	E-22
TIBPAL20R4	P20R4	E-23
TIBPAL20R6	P20R6	E-24
TIBPAL20R8	P20R8	E-25
TIBPAL22V10	P22V10	E-26
TIBPAL22VP10	P22VP10	E-28
TIB82S105B	A105B	E-3
TIB82S167B	A167B	E-5
TIBPLS506	A506	E-7
TIBPSG507	A507	E-11
TICPAL16L8	P16L8	E-15
TICPAL16R4	P16R4	E-17
TICPAL16R6	P16R6	E-18
TICPAL16R8	P16R8	E-19
TICPAL22V10	P22V10	E-26

Appendix E

Logic Diagrams



a105b-1

DEFAULT STATES

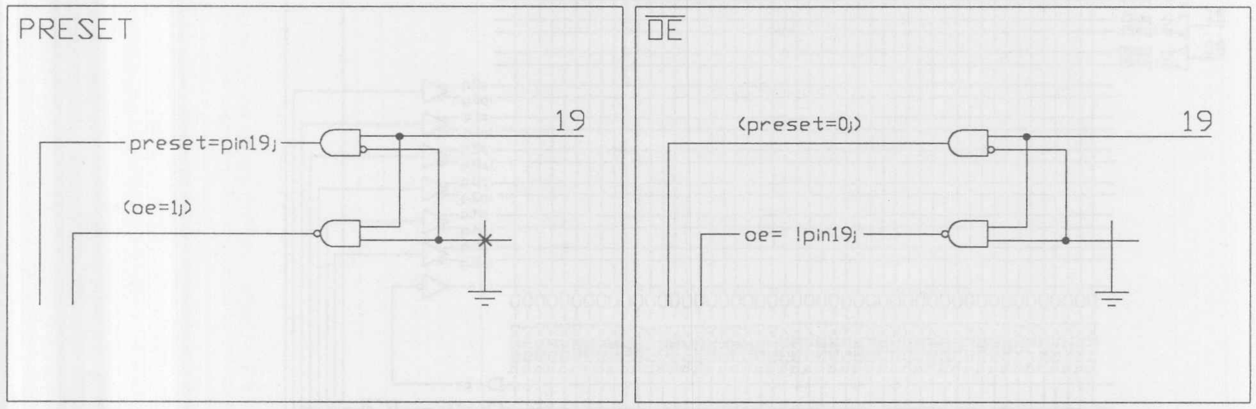
All unreferenced gates remain unprogrammed.

Unreferenced AND gates default to logic 0. Unreferenced OR gates are undefined.

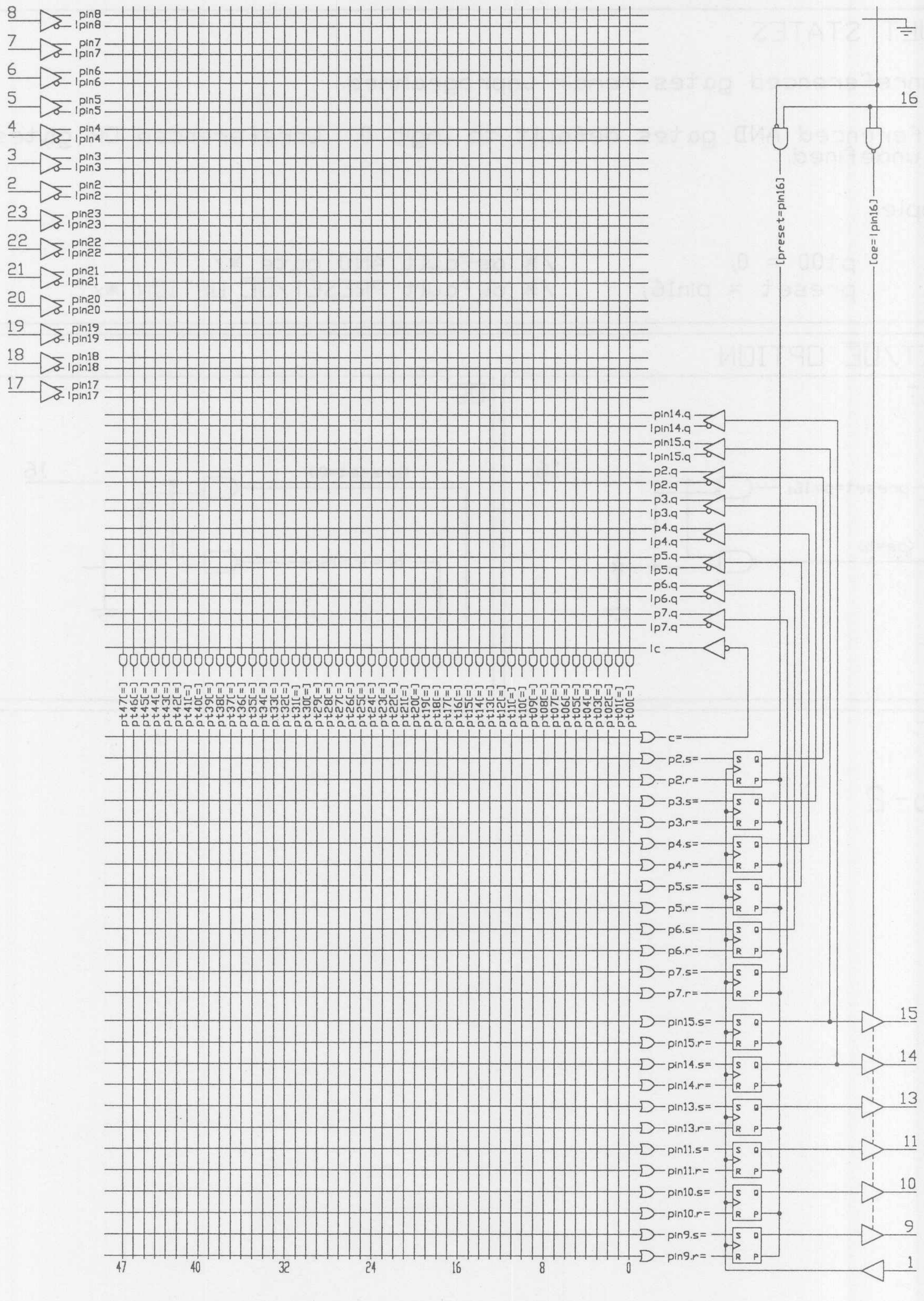
Example:

```
        pt00 = 0;              /* default AND gate */
        preset = pin19;        /* default PRESET/OE OPTION */
```

PRESET/$\overline{\text{OE}}$ OPTION



a105b-2



8
pin8
!pin8
7
pin7
!pin7
6
pin6
!pin6
5
pin5
!pin5
4
pin4
!pin4
3
pin3
!pin3
2
pin2
!pin2
23
pin23
!pin23
22
pin22
!pin22
21
pin21
!pin21
20
pin20
!pin20
19
pin19
!pin19
18
pin18
!pin18
17
pin17
!pin17
16
[preset=pin16]
[oe=!pin16]
pin14.q
!pin14.q
pin15.q
!pin15.q
p2.q
!p2.q
p3.q
!p3.q
p4.q
!p4.q
p5.q
!p5.q
p6.q
!p6.q
p7.q
!p7.q
!c
pt47[=]
pt46[=]
pt45[=]
pt44[=]
pt43[=]
pt42[=]
pt41[=]
pt40[=]
pt39[=]
pt38[=]
pt37[=]
pt36[=]
pt35[=]
pt34[=]
pt33[=]
pt32[=]
pt31[=]
pt30[=]
pt29[=]
pt28[=]
pt27[=]
pt26[=]
pt25[=]
pt24[=]
pt23[=]
pt22[=]
pt21[=]
pt20[=]
pt19[=]
pt18[=]
pt17[=]
pt16[=]
pt15[=]
pt14[=]
pt13[=]
pt12[=]
pt11[=]
pt10[=]
pt09[=]
pt08[=]
pt07[=]
pt06[=]
pt05[=]
pt04[=]
pt03[=]
pt02[=]
pt01[=]
pt00[=]
c=
p2.s=
p2.r=
p3.s=
p3.r=
p4.s=
p4.r=
p5.s=
p5.r=
p6.s=
p6.r=
p7.s=
p7.r=
pin15.s=
pin15.r=
pin14.s=
pin14.r=
pin13.s=
pin13.r=
pin11.s=
pin11.r=
pin10.s=
pin10.r=
pin9.s=
pin9.r=
S Q
R P
15
14
13
11
10
9
1
47
40
32
24
16
8
0

a167b-1

DEFAULT STATES

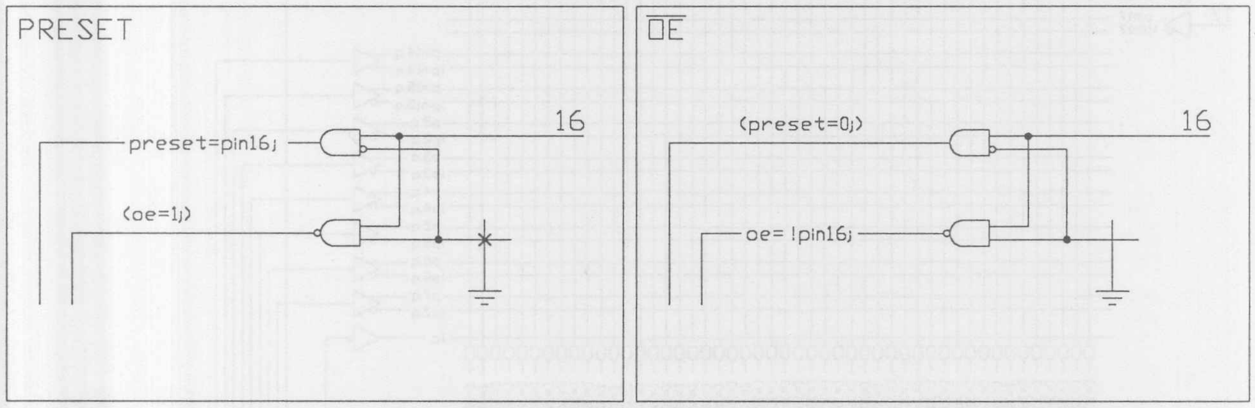
All unreferenced gates remain unprogrammed.

Unreferenced AND gates default to logic 0. Unreferenced OR gates are undefined.

Example:

```
        pt00 = 0;              /* default AND gate */
        preset = pin16;        /* default PRESET/OE OPTION */
```

PRESET/$\overline{\text{OE}}$ OPTION



a167b-2

DEFAULT STATES

All unreferenced gates remain unprogrammed.

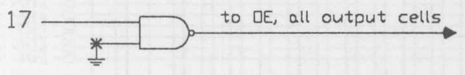
Unreferenced AND gates default to logic 0. Unreferenced OR gates are undefined. The Output Cells default to registered operation. Registers are positive-edge triggered. Outputs are permanently enabled.

Example:

```
pt96 = 0;           /* default AND gate */
oe = 1;             /* output buffers always enabled */
clk = pin1;         /* registers positive-edge triggered */
```

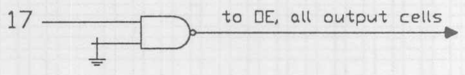
OUTPUT ENABLE

OUTPUTS PERMANENTLY ENABLED



```
oe = 1;        /* default */
```

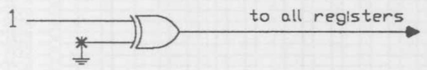
PIN 17 OUTPUT ENABLE



```
oe = !pin17;
```

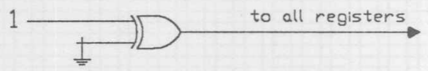
CLOCK POLARITY

POSITIVE-EDGE TRIGGERED



```
clk = pin1;      /* default */
```

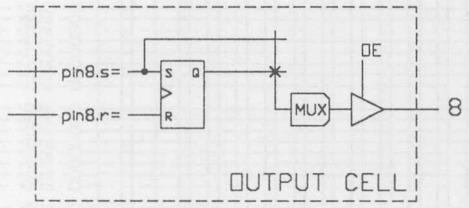
NEGATIVE-EDGE TRIGGERED



```
clk = !pin1;
```

TYPICAL OUTPUT CELL OPERATION

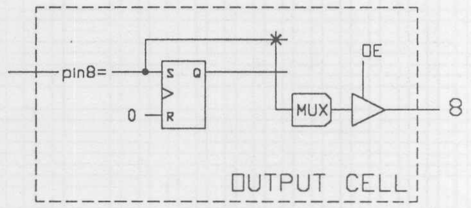
REGISTERED



No signal specification for pin8= is permitted.

Unprogrammed State

COMBINATORIAL

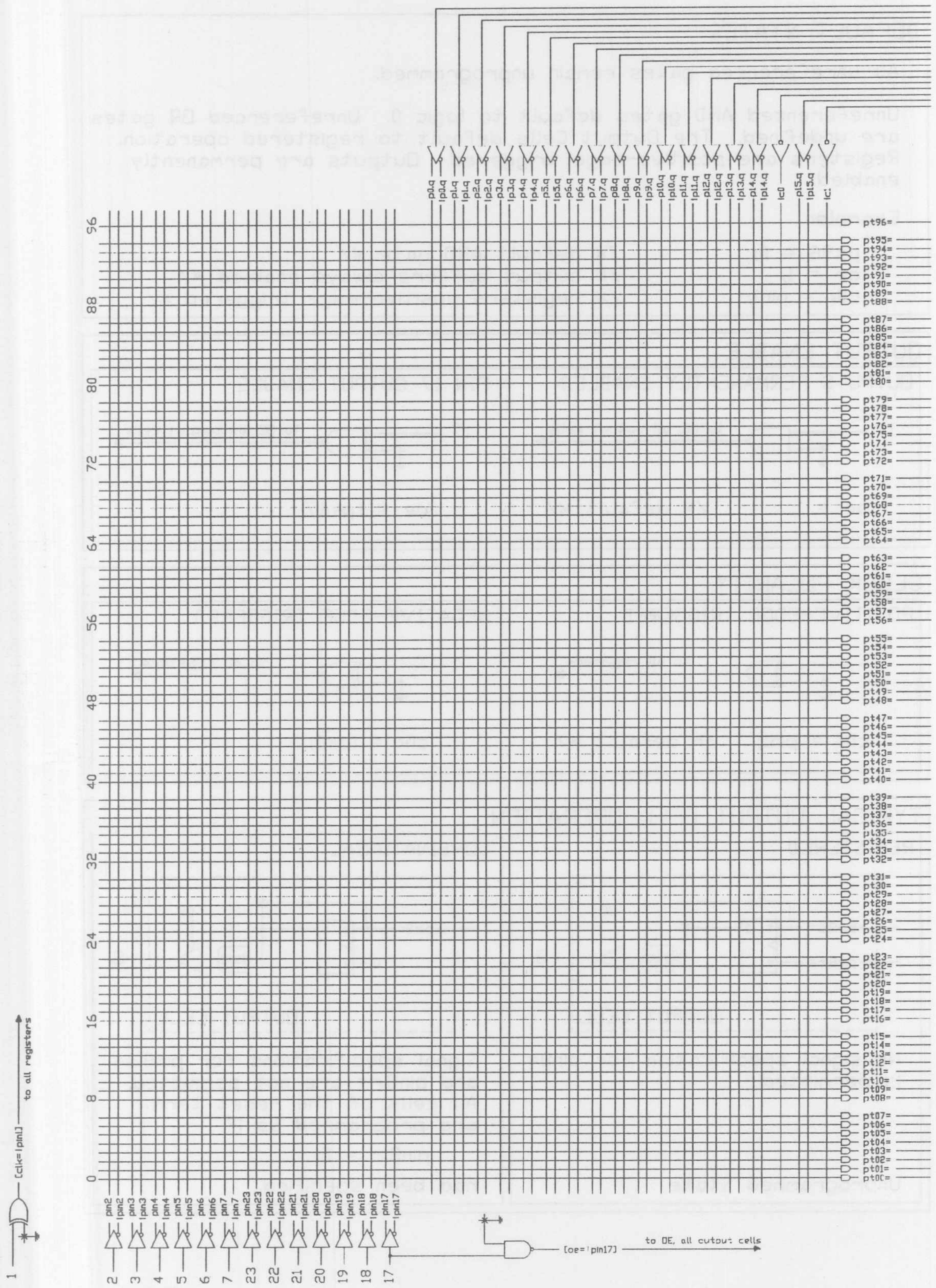


Signal specifications for pin8.s= and pin8.r= are not permitted. All cells of the reset term are programmed as if

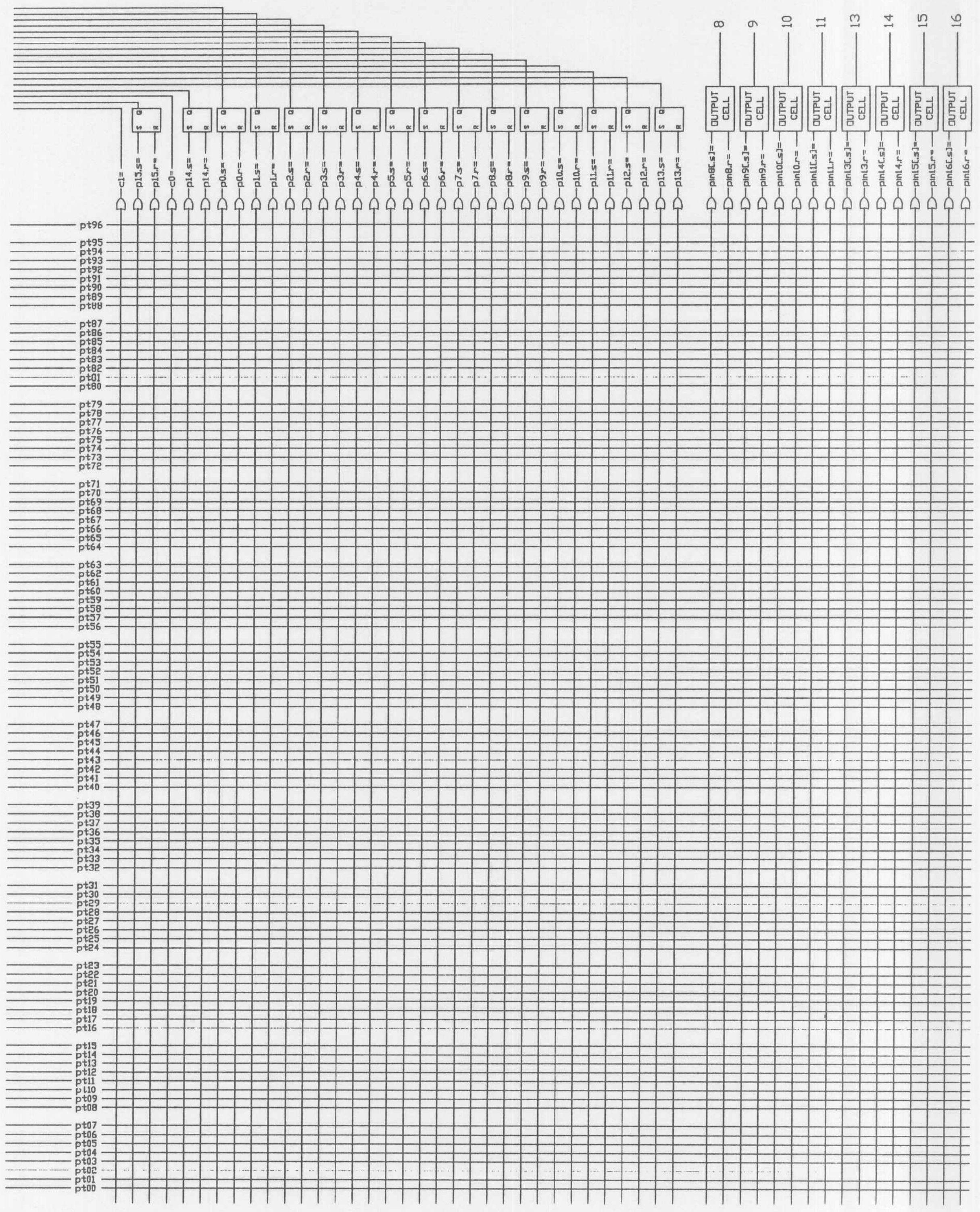
pin8.r = 0;

had been specified.

a506-1



a506-2


8
9
10
11
13
14
15
16
OUTPUT CELL
S Q R
c1=
p15.s=
p15.r=
c0=
p14.s=
p14.r=
p0.s=
p0.r=
p1.s=
p1.r=
p2.s=
p2.r=
p3.s=
p3.r=
p4.s=
p4.r=
p5.s=
p5.r=
p6.s=
p6.r=
p7.s=
p7.r=
p8.s=
p8.r=
p9.s=
p9.r=
p10.s=
p10.r=
p11.s=
p11.r=
p12.s=
p12.r=
p13.s=
p13.r=
pin8[.s]=
pin8.r=
pin9[.s]=
pin9.r=
pin10[.s]=
pin10.r=
pin11[.s]=
pin11.r=
pin13[.s]=
pin13.r=
pin14[.s]=
pin14.r=
pin15[.s]=
pin15.r=
pin16[.s]=
pin16.r=
pt96
pt95
pt94
pt93
pt92
pt91
pt90
pt89
pt88
pt87
pt86
pt85
pt84
pt83
pt82
pt81
pt80
pt79
pt78
pt77
pt76
pt75
pt74
pt73
pt72
pt71
pt70
pt69
pt68
pt67
pt66
pt65
pt64
pt63
pt62
pt61
pt60
pt59
pt58
pt57
pt56
pt55
pt54
pt53
pt52
pt51
pt50
pt49
pt48
pt47
pt46
pt45
pt44
pt43
pt42
pt41
pt40
pt39
pt38
pt37
pt36
pt35
pt34
pt33
pt32
pt31
pt30
pt29
pt28
pt27
pt26
pt25
pt24
pt23
pt22
pt21
pt20
pt19
pt18
pt17
pt16
pt15
pt14
pt13
pt12
pt11
pt10
pt09
pt08
pt07
pt06
pt05
pt04
pt03
pt02
pt01
pt00

a506-3

DEFAULT STATES

All unreferenced gates remain unprogrammed.

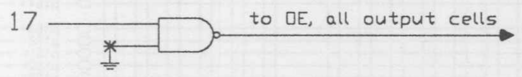
Unreferenced AND gates default to logic 0. Unreferenced OR gates are undefined. The Output Cells default to registered operation. Registers are positive-edge triggered. Outputs are premanently enabled.

Example:

```
pt79 = 0;          /* default AND gate */
oe = 1;            /* output buffers always enabled */
clk = pin1;        /* registers positive-edge triggered */
```

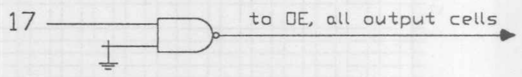
OUTPUT ENABLE

OUTPUTS PERMANENTLY ENABLED



oe = 1;   /* default */

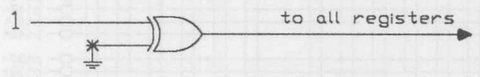
PIN 17 OUTPUT ENABLE



oe = !pin17;

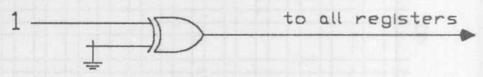
CLOCK POLARITY

POSITIVE-EDGE TRIGGERED



clk = pin1;   /* default */

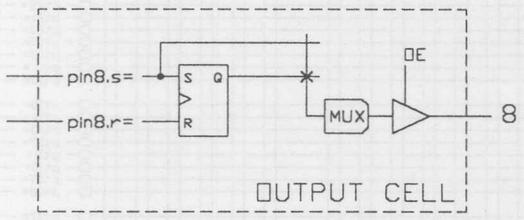
NEGATIVE-EDGE TRIGGERED



clk = !pin1;

TYPICAL OUTPUT CELL OPERATION

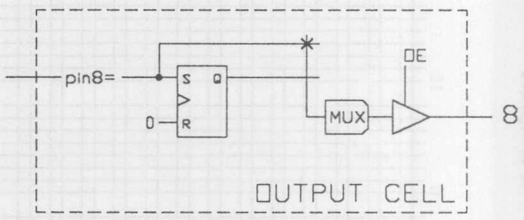
REGISTERED



No signal specification for pin8= is permitted.

Unprogrammed State

COMBINATORIAL

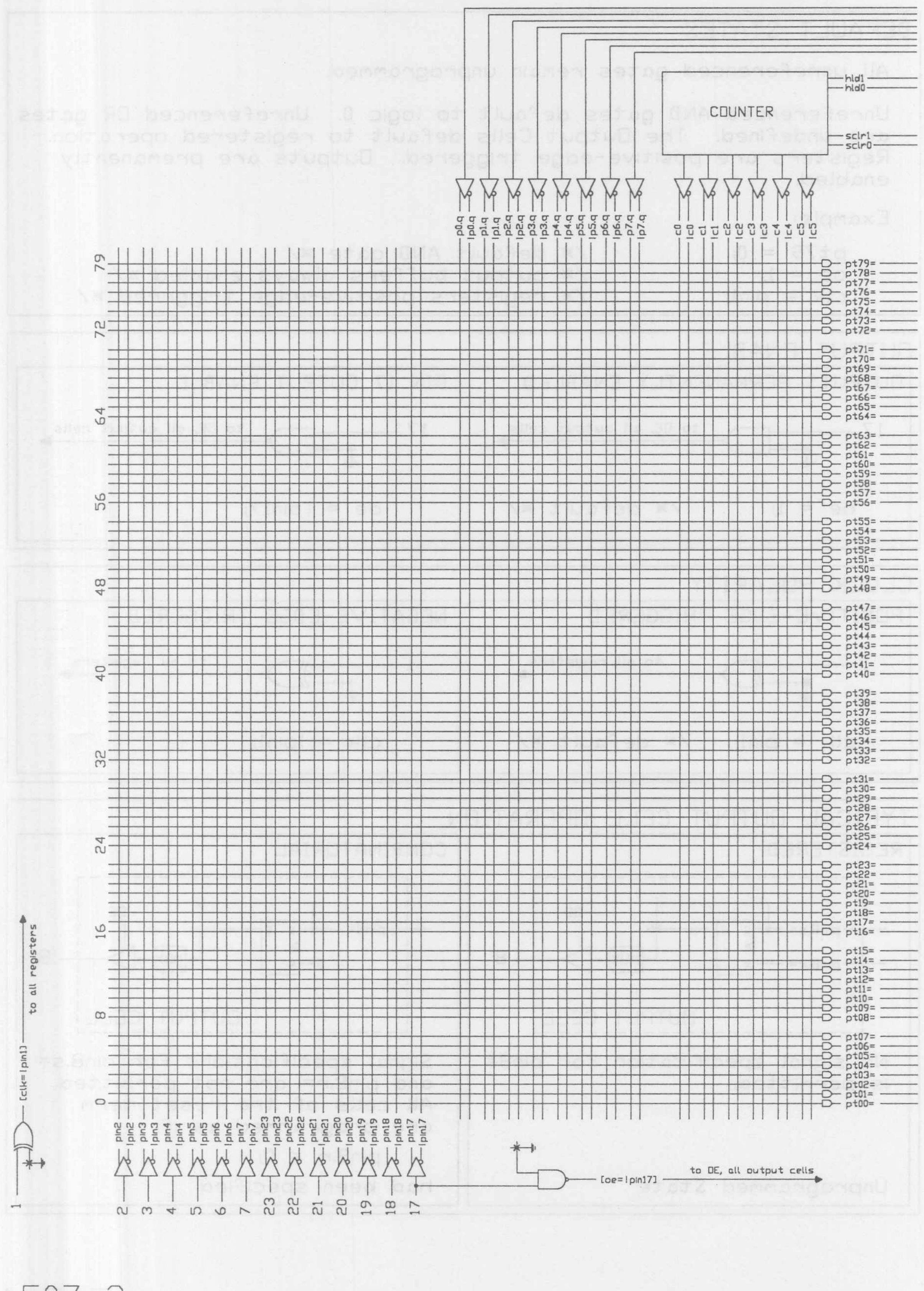


Signal specifications for pin8.s= and pin8.r= are not permitted. All cells of the reset term are programmed as if

pin8.r = 0;

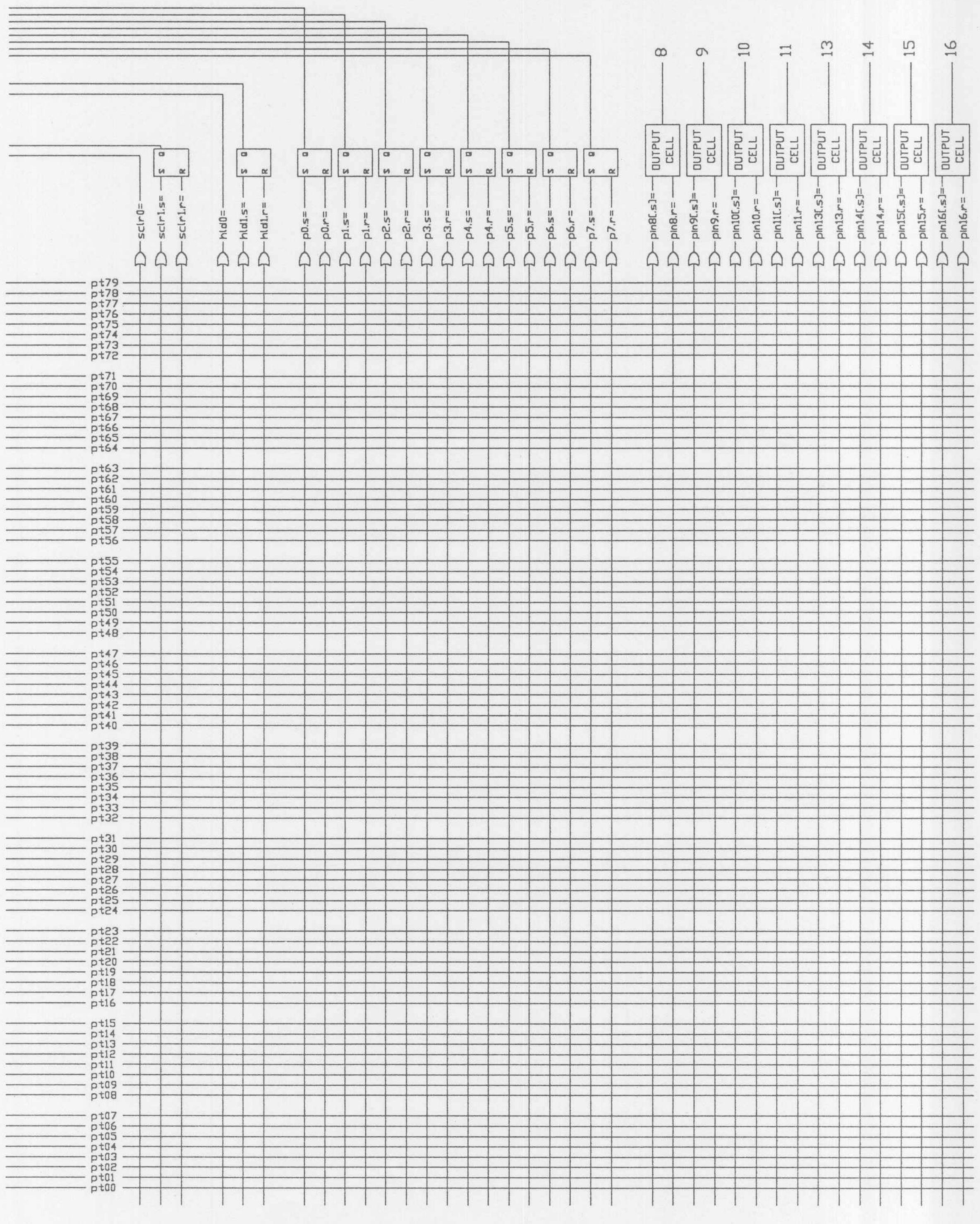
had been specified.

a507-1



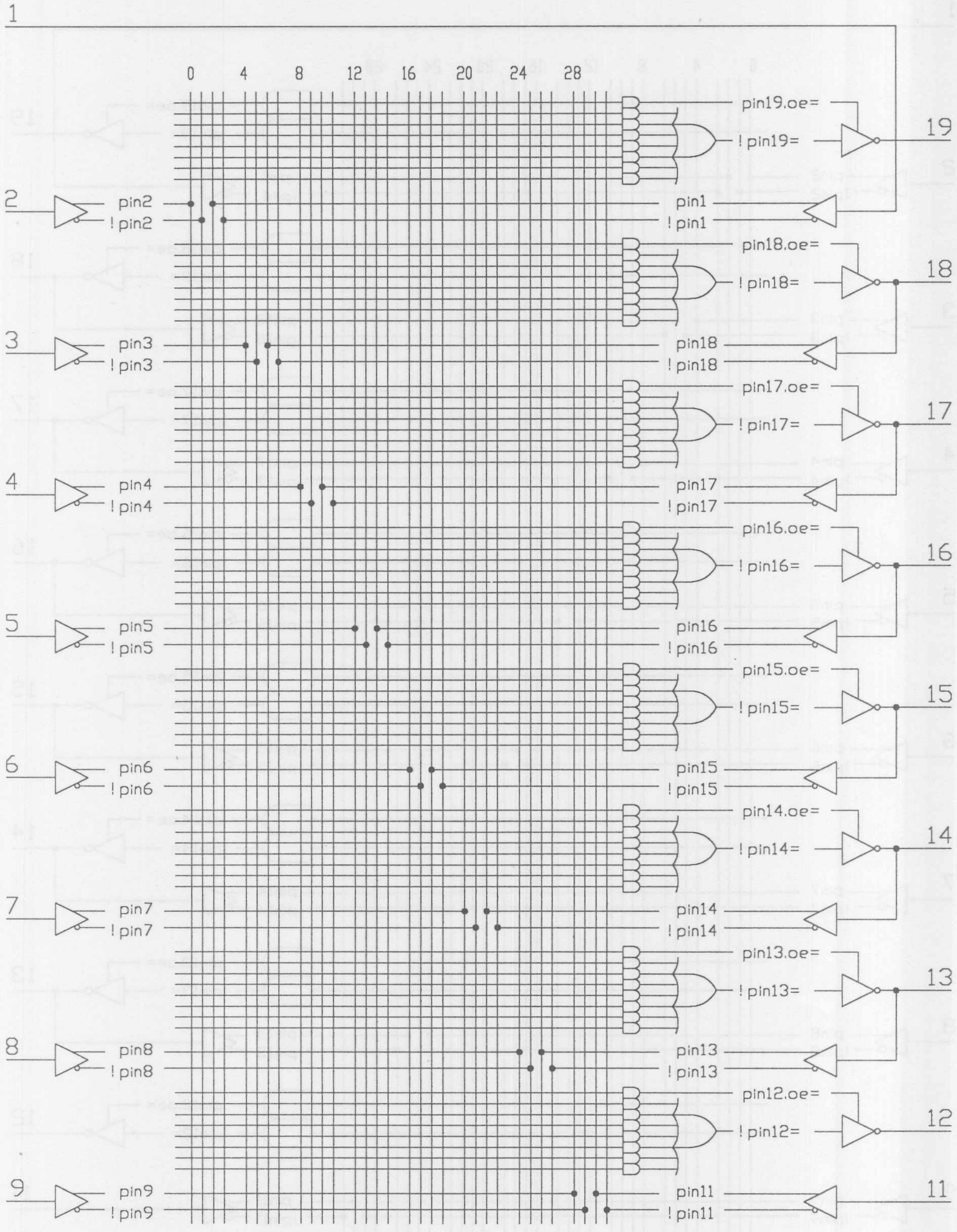
COUNTER
hld1
hld0
sclr1
sclr0
p0.q
!p0.q
p1.q
!p1.q
p2.q
!p2.q
p3.q
!p3.q
p4.q
!p4.q
p5.q
!p5.q
p6.q
!p6.q
p7.q
!p7.q
c0
!c0
c1
!c1
c2
!c2
c3
!c3
c4
!c4
c5
!c5
79
72
64
56
48
40
32
24
16
8
0
pt79=
pt78=
pt77=
pt76=
pt75=
pt74=
pt73=
pt72=
pt71=
pt70=
pt69=
pt68=
pt67=
pt66=
pt65=
pt64=
pt63=
pt62=
pt61=
pt60=
pt59=
pt58=
pt57=
pt56=
pt55=
pt54=
pt53=
pt52=
pt51=
pt50=
pt49=
pt48=
pt47=
pt46=
pt45=
pt44=
pt43=
pt42=
pt41=
pt40=
pt39=
pt38=
pt37=
pt36=
pt35=
pt34=
pt33=
pt32=
pt31=
pt30=
pt29=
pt28=
pt27=
pt26=
pt25=
pt24=
pt23=
pt22=
pt21=
pt20=
pt19=
pt18=
pt17=
pt16=
pt15=
pt14=
pt13=
pt12=
pt11=
pt10=
pt09=
pt08=
pt07=
pt06=
pt05=
pt04=
pt03=
pt02=
pt01=
pt00=
[clk=!pin1]
to all registers
pin2
!pin2
pin3
!pin3
pin4
!pin4
pin5
!pin5
pin6
!pin6
pin7
!pin7
pin23
!pin23
pin22
!pin22
pin21
!pin21
pin20
!pin20
pin19
!pin19
pin18
!pin18
pin17
!pin17
1
2
3
4
5
6
7
23
22
21
20
19
18
17
[oe=!pin17]
to OE, all output cells

a507-2



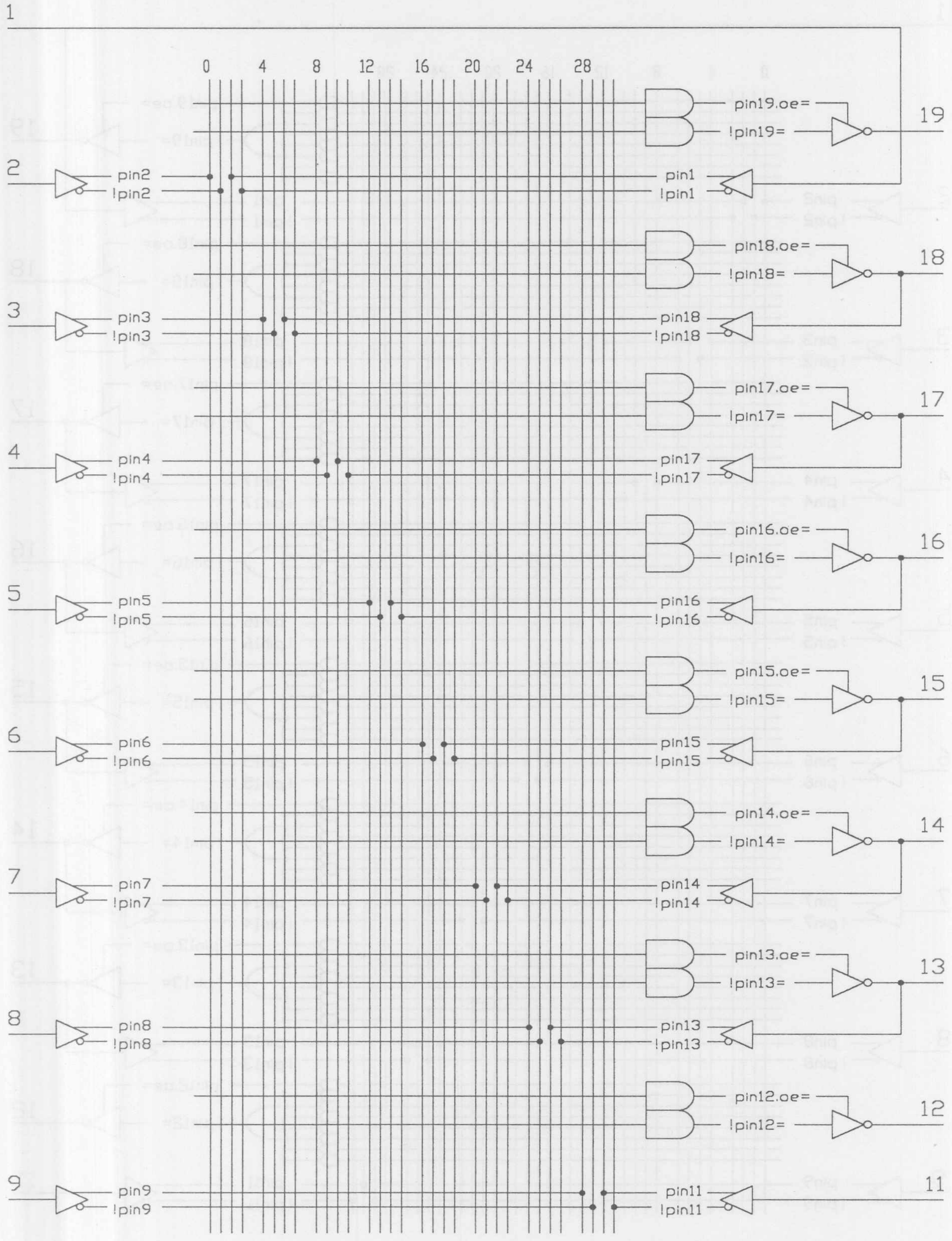
8
9
10
11
13
14
15
16
OUTPUT CELL
S Q R
sclr0=
sclr1.s=
sclr1.r=
hld0=
hld1.s=
hld1.r=
p0.s=
p0.r=
p1.s=
p1.r=
p2.s=
p2.r=
p3.s=
p3.r=
p4.s=
p4.r=
p5.s=
p5.r=
p6.s=
p6.r=
p7.s=
p7.r=
pin8[.s]=
pin8.r=
pin9[.s]=
pin9.r=
pin10[.s]=
pin10.r=
pin11[.s]=
pin11.r=
pin13[.s]=
pin13.r=
pin14[.s]=
pin14.r=
pin15[.s]=
pin15.r=
pin16[.s]=
pin16.r=
pt79
pt78
pt77
pt76
pt75
pt74
pt73
pt72
pt71
pt70
pt69
pt68
pt67
pt66
pt65
pt64
pt63
pt62
pt61
pt60
pt59
pt58
pt57
pt56
pt55
pt54
pt53
pt52
pt51
pt50
pt49
pt48
pt47
pt46
pt45
pt44
pt43
pt42
pt41
pt40
pt39
pt38
pt37
pt36
pt35
pt34
pt33
pt32
pt31
pt30
pt29
pt28
pt27
pt26
pt25
pt24
pt23
pt22
pt21
pt20
pt19
pt18
pt17
pt16
pt15
pt14
pt13
pt12
pt11
pt10
pt09
pt08
pt07
pt06
pt05
pt04
pt03
pt02
pt01
pt00

a507-3



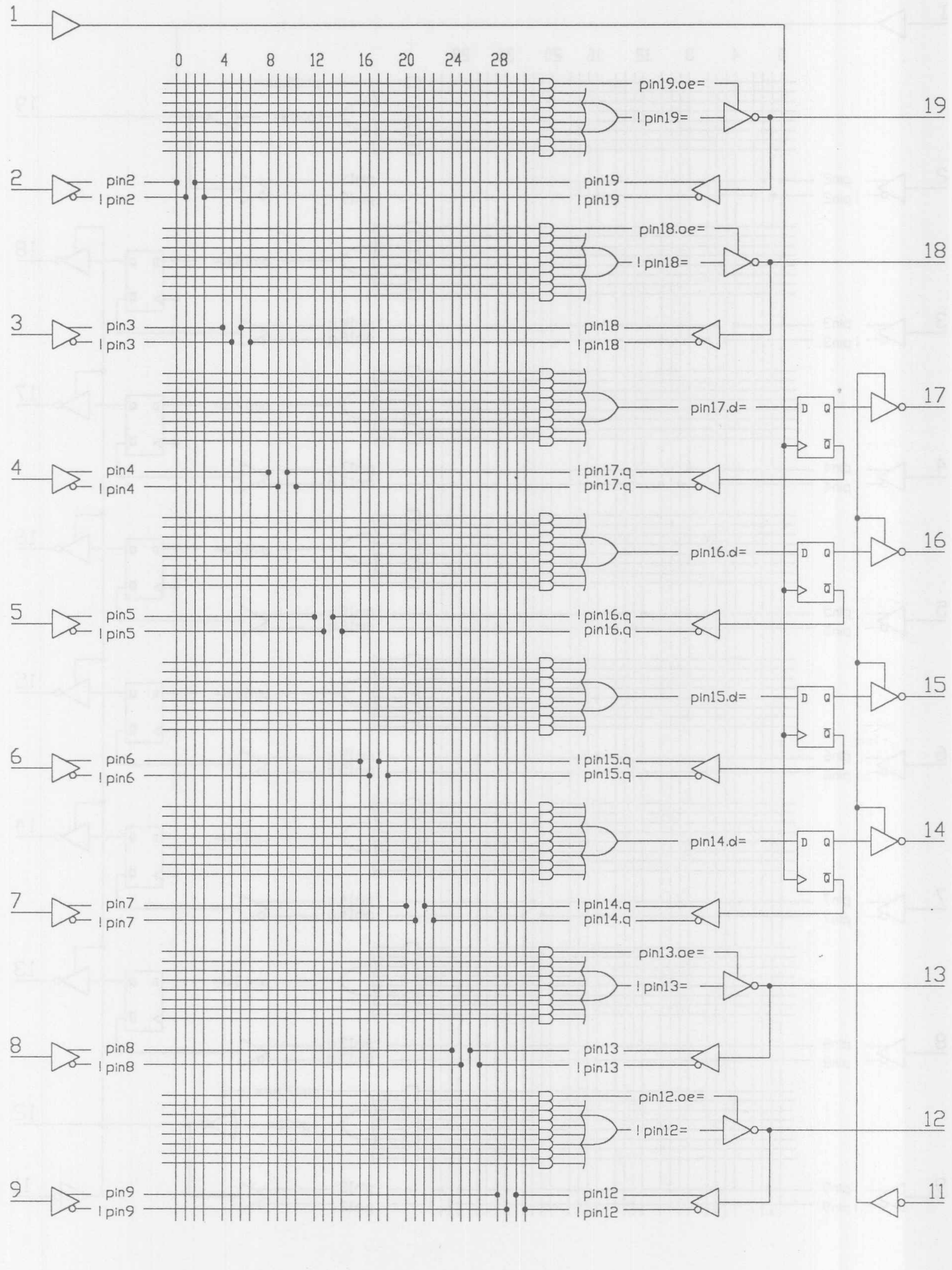
1
0
4
8
12
16
20
24
28
pin19.oe=
! pin19=
19
2
pin2
! pin2
pin1
! pin1
pin18.oe=
! pin18=
18
3
pin3
! pin3
pin18
! pin18
pin17.oe=
! pin17=
17
4
pin4
! pin4
pin17
! pin17
pin16.oe=
! pin16=
16
5
pin5
! pin5
pin16
! pin16
pin15.oe=
! pin15=
15
6
pin6
! pin6
pin15
! pin15
pin14.oe=
! pin14=
14
7
pin7
! pin7
pin14
! pin14
pin13.oe=
! pin13=
13
8
pin8
! pin8
pin13
! pin13
pin12.oe=
! pin12=
12
9
pin9
! pin9
pin11
! pin11
11

p16l8



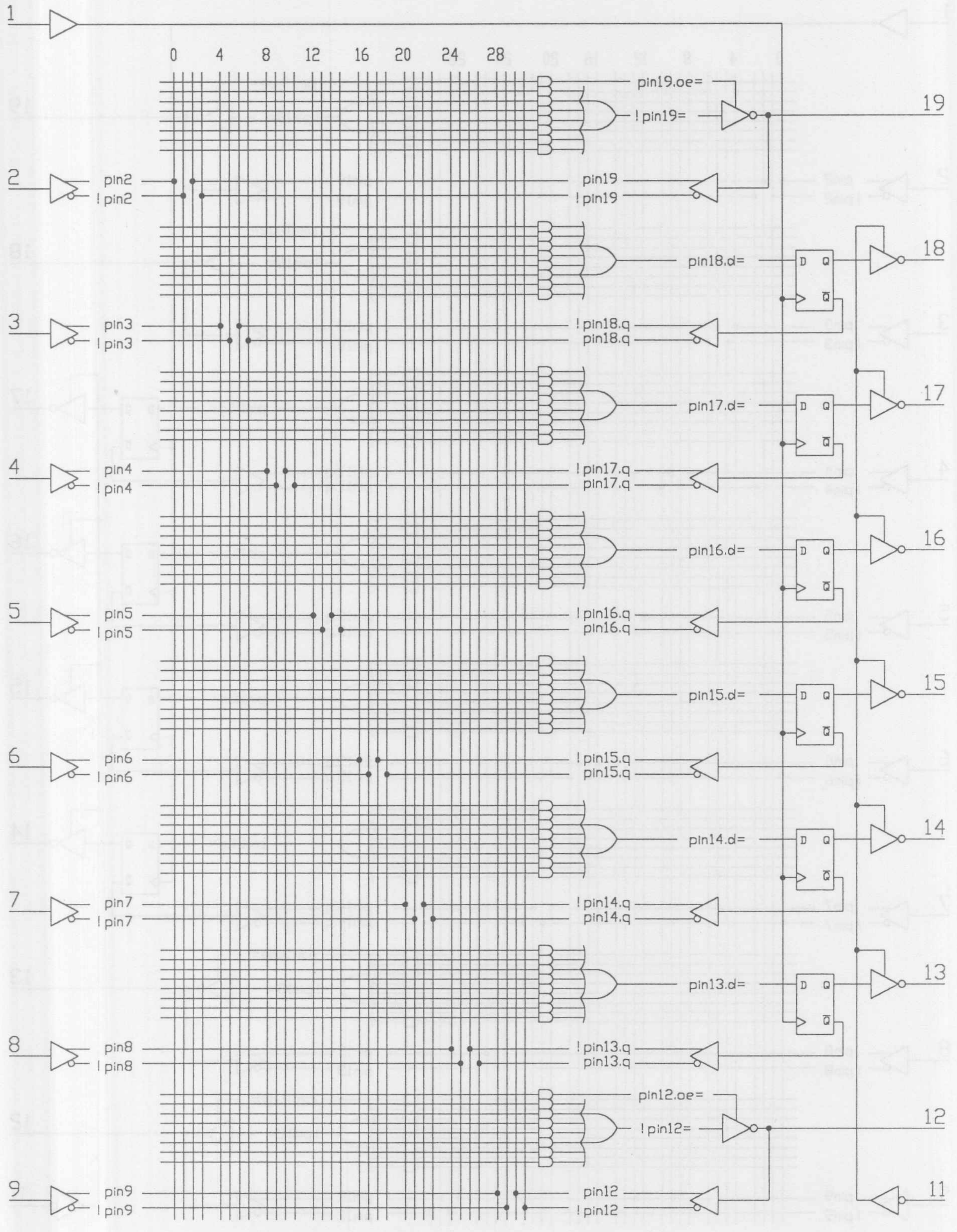
1
0
4
8
12
16
20
24
28
2
pin2
!pin2
3
pin3
!pin3
4
pin4
!pin4
5
pin5
!pin5
6
pin6
!pin6
7
pin7
!pin7
8
pin8
!pin8
9
pin9
!pin9
pin19.oe=
!pin19=
19
pin1
!pin1
pin18.oe=
!pin18=
18
pin18
!pin18
pin17.oe=
!pin17=
17
pin17
!pin17
pin16.oe=
!pin16=
16
pin16
!pin16
pin15.oe=
!pin15=
15
pin15
!pin15
pin14.oe=
!pin14=
14
pin14
!pin14
pin13.oe=
!pin13=
13
pin13
!pin13
pin12.oe=
!pin12=
12
pin11
!pin11
11

p16n8

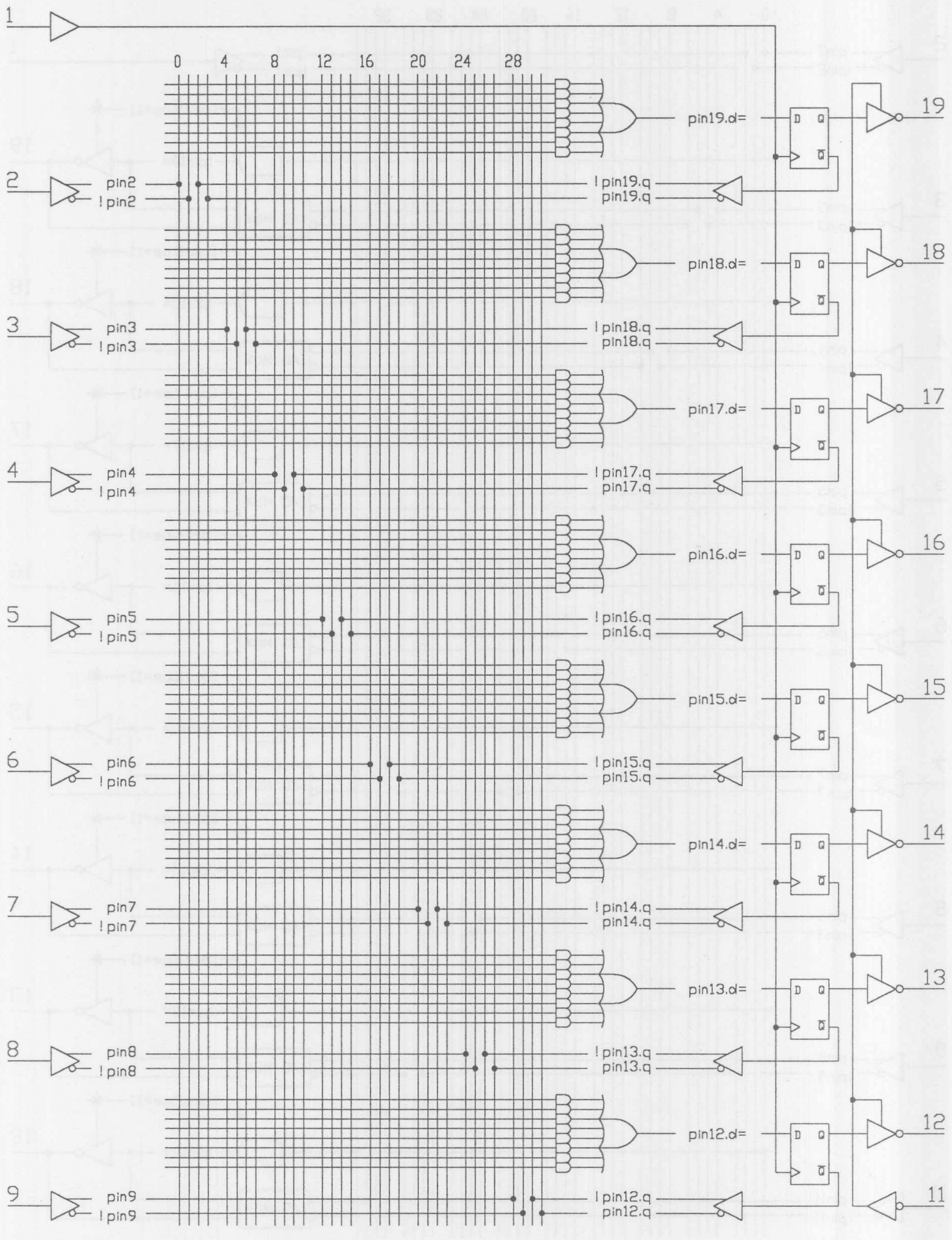


1
0
4
8
12
16
20
24
28
pin19.oe=
!pin19=
19
2
pin2
!pin2
pin19
!pin19
pin18.oe=
!pin18=
18
3
pin3
!pin3
pin18
!pin18
pin17.d=
17
4
pin4
!pin4
!pin17.q
pin17.q
pin16.d=
16
5
pin5
!pin5
!pin16.q
pin16.q
pin15.d=
15
6
pin6
!pin6
!pin15.q
pin15.q
pin14.d=
14
7
pin7
!pin7
!pin14.q
pin14.q
pin13.oe=
!pin13=
13
8
pin8
!pin8
pin13
!pin13
pin12.oe=
!pin12=
12
9
pin9
!pin9
pin12
!pin12
11
D
Q

p16r4

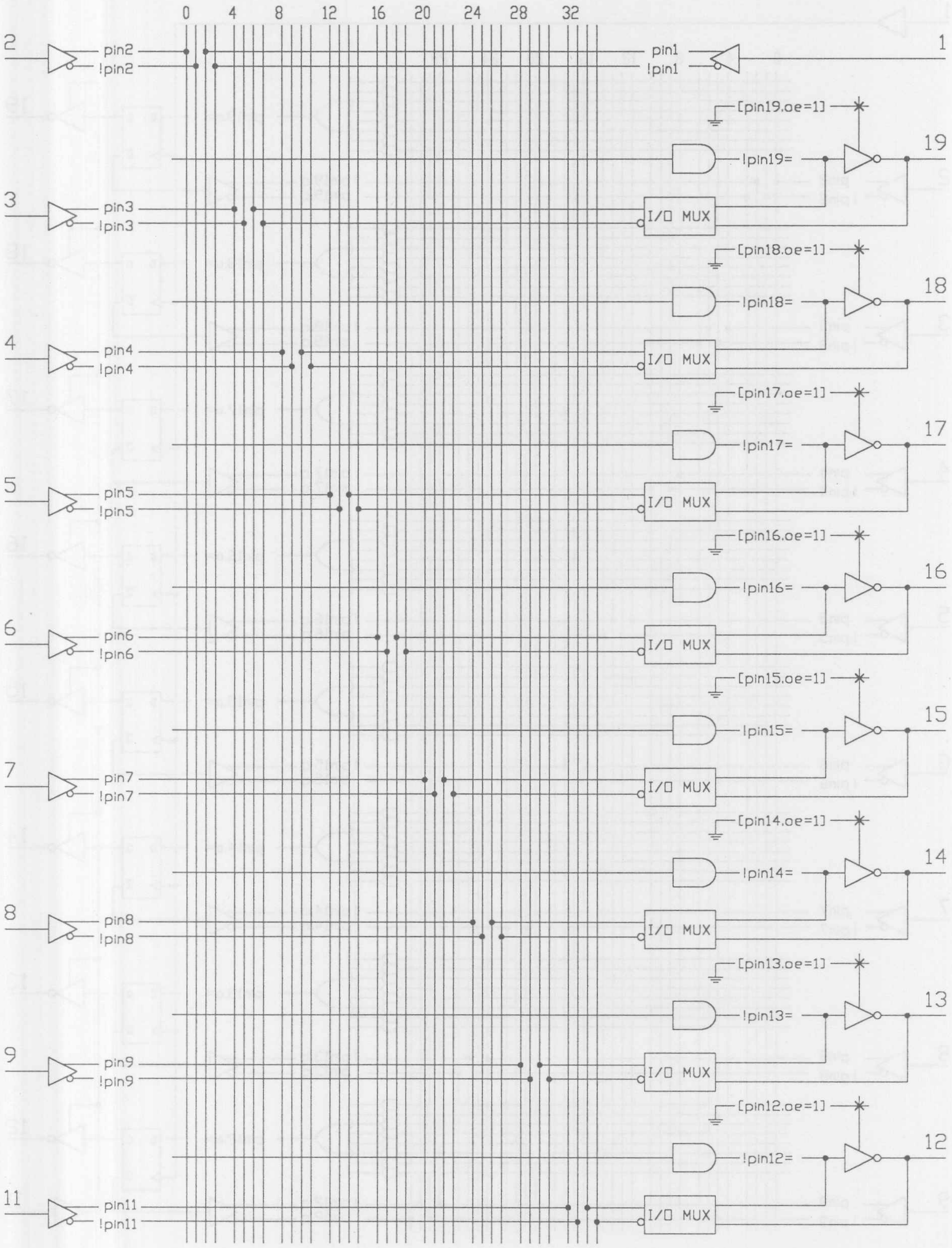

1
0
4
8
12
16
20
24
28
pin19.oe=
!pin19=
19
2
pin2
!pin2
pin19
!pin19
pin18.d=
18
3
pin3
!pin3
!pin18.q
pin18.q
pin17.d=
17
4
pin4
!pin4
!pin17.q
pin17.q
pin16.d=
16
5
pin5
!pin5
!pin16.q
pin16.q
pin15.d=
15
6
pin6
!pin6
!pin15.q
pin15.q
pin14.d=
14
7
pin7
!pin7
!pin14.q
pin14.q
pin13.d=
13
8
pin8
!pin8
!pin13.q
pin13.q
pin12.oe=
!pin12=
12
9
pin9
!pin9
pin12
!pin12
11
D
Q

p16r6

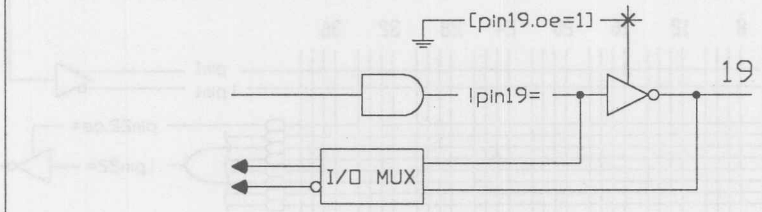


1
2
3
4
5
6
7
8
9
0
4
8
12
16
20
24
28
pin2
!pin2
pin3
!pin3
pin4
!pin4
pin5
!pin5
pin6
!pin6
pin7
!pin7
pin8
!pin8
pin9
!pin9
pin19.d=
!pin19.q
pin19.q
pin18.d=
!pin18.q
pin18.q
pin17.d=
!pin17.q
pin17.q
pin16.d=
!pin16.q
pin16.q
pin15.d=
!pin15.q
pin15.q
pin14.d=
!pin14.q
pin14.q
pin13.d=
!pin13.q
pin13.q
pin12.d=
!pin12.q
pin12.q
D
Q
19
18
17
16
15
14
13
12
11

p16r8


0
4
8
12
16
20
24
28
32
2
pin2
!pin2
pin1
!pin1
1
[pin19.oe=1]
!pin19=
19
3
pin3
!pin3
I/O MUX
[pin18.oe=1]
!pin18=
18
4
pin4
!pin4
I/O MUX
[pin17.oe=1]
!pin17=
17
5
pin5
!pin5
I/O MUX
[pin16.oe=1]
!pin16=
16
6
pin6
!pin6
I/O MUX
[pin15.oe=1]
!pin15=
15
7
pin7
!pin7
I/O MUX
[pin14.oe=1]
!pin14=
14
8
pin8
!pin8
I/O MUX
[pin13.oe=1]
!pin13=
13
9
pin9
!pin9
I/O MUX
[pin12.oe=1]
!pin12=
12
11
pin11
!pin11
I/O MUX

p18n8-1



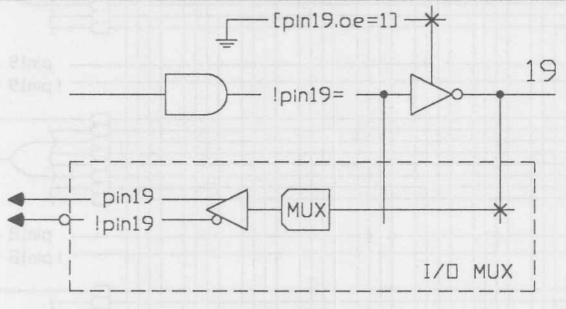
If there is no signal specification for pin19.oe the Architectural Fuse remains intact and the output buffer is in high-impedence state.

If there is a signal specification of

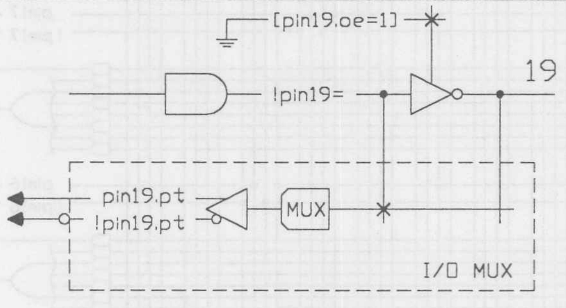
pin19.oe = 1;

the fuse is programmed and the output buffer is in output state.

TYPICAL OUTPUT BUFFER PROGRAMMING



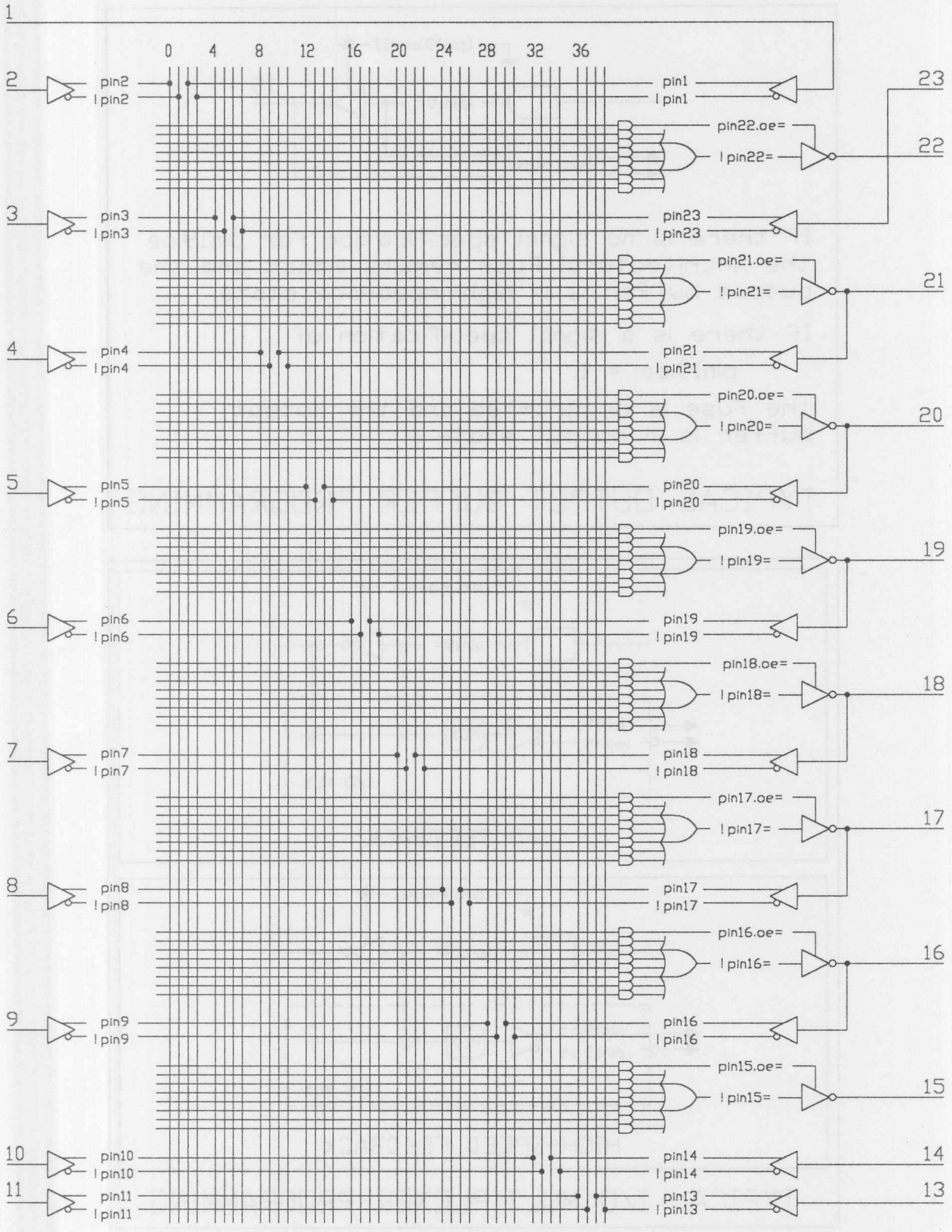
I/O FEEDBACK



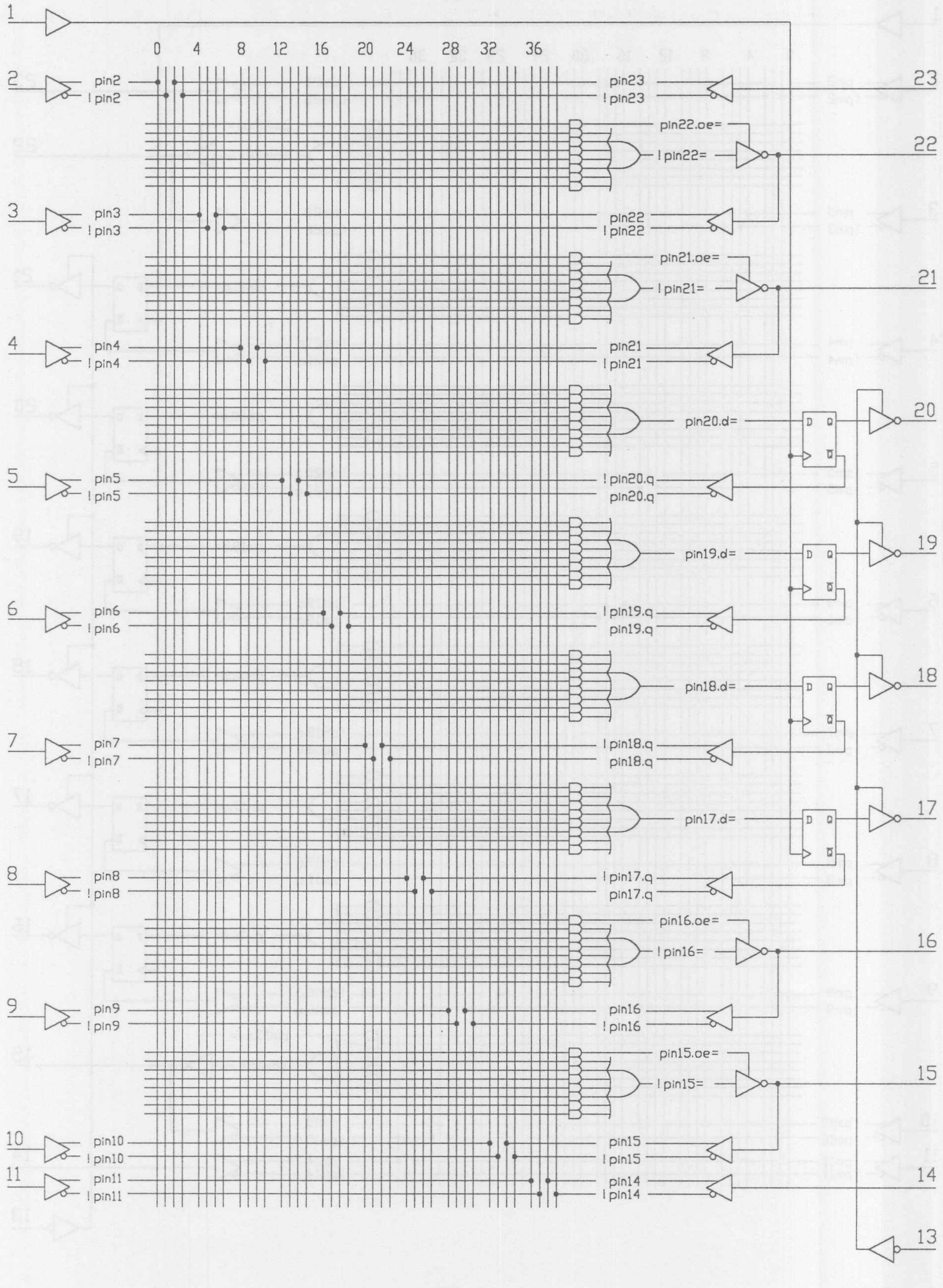
HIGH-SPEED FEEDBACK

TYPICAL I/O MULTIPLEXER PROGRAMMING

p18n8-2

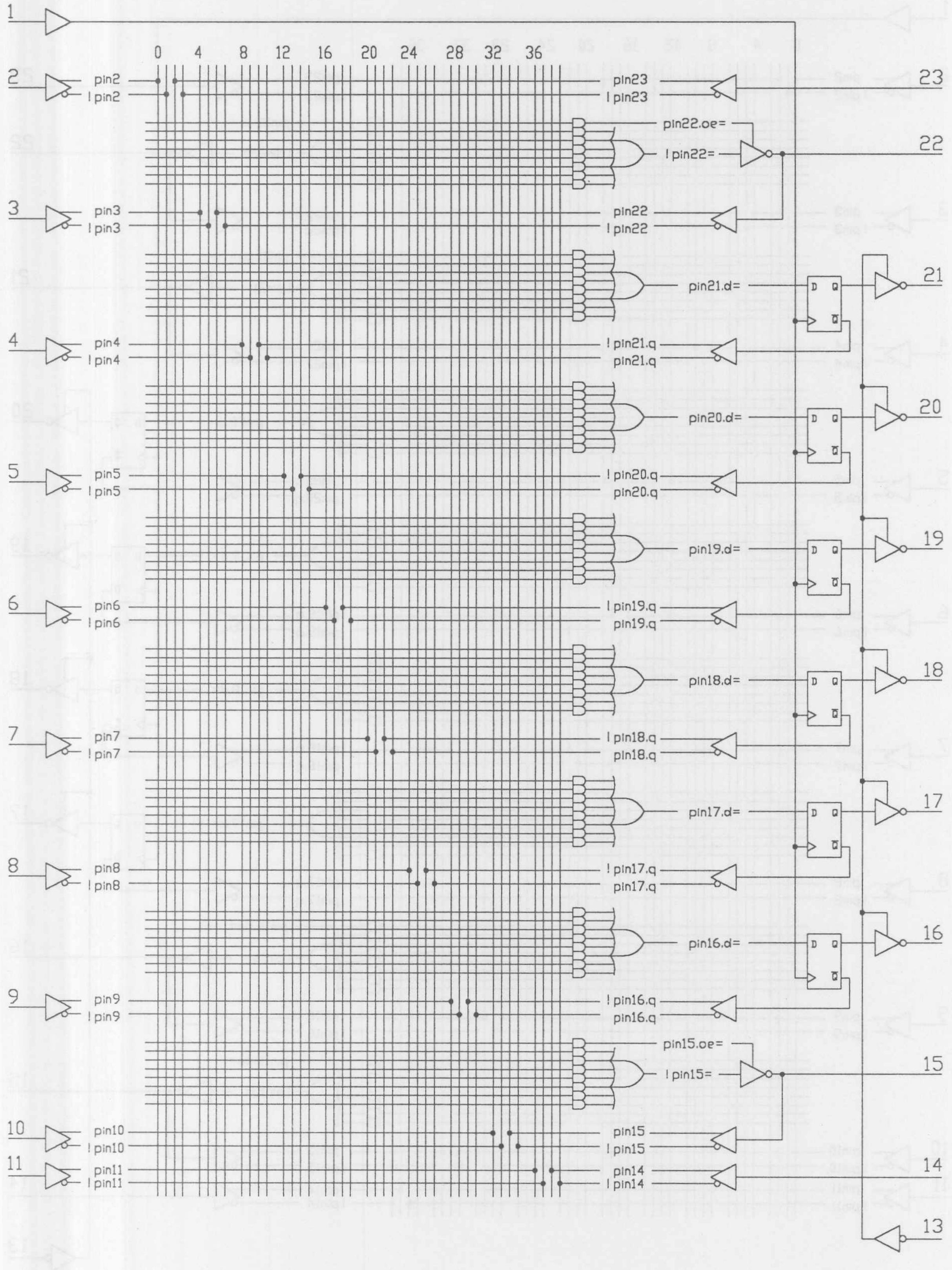

1
0
4
8
12
16
20
24
28
32
36
2
pin2
! pin2
pin1
! pin1
23
pin22.oe=
! pin22=
22
3
pin3
! pin3
pin23
! pin23
pin21.oe=
! pin21=
21
4
pin4
! pin4
pin21
! pin21
pin20.oe=
! pin20=
20
5
pin5
! pin5
pin20
! pin20
pin19.oe=
! pin19=
19
6
pin6
! pin6
pin19
! pin19
pin18.oe=
! pin18=
18
7
pin7
! pin7
pin18
! pin18
pin17.oe=
! pin17=
17
8
pin8
! pin8
pin17
! pin17
pin16.oe=
! pin16=
16
9
pin9
! pin9
pin16
! pin16
pin15.oe=
! pin15=
15
10
pin10
! pin10
pin14
! pin14
14
11
pin11
! pin11
pin13
! pin13
13

p20l8



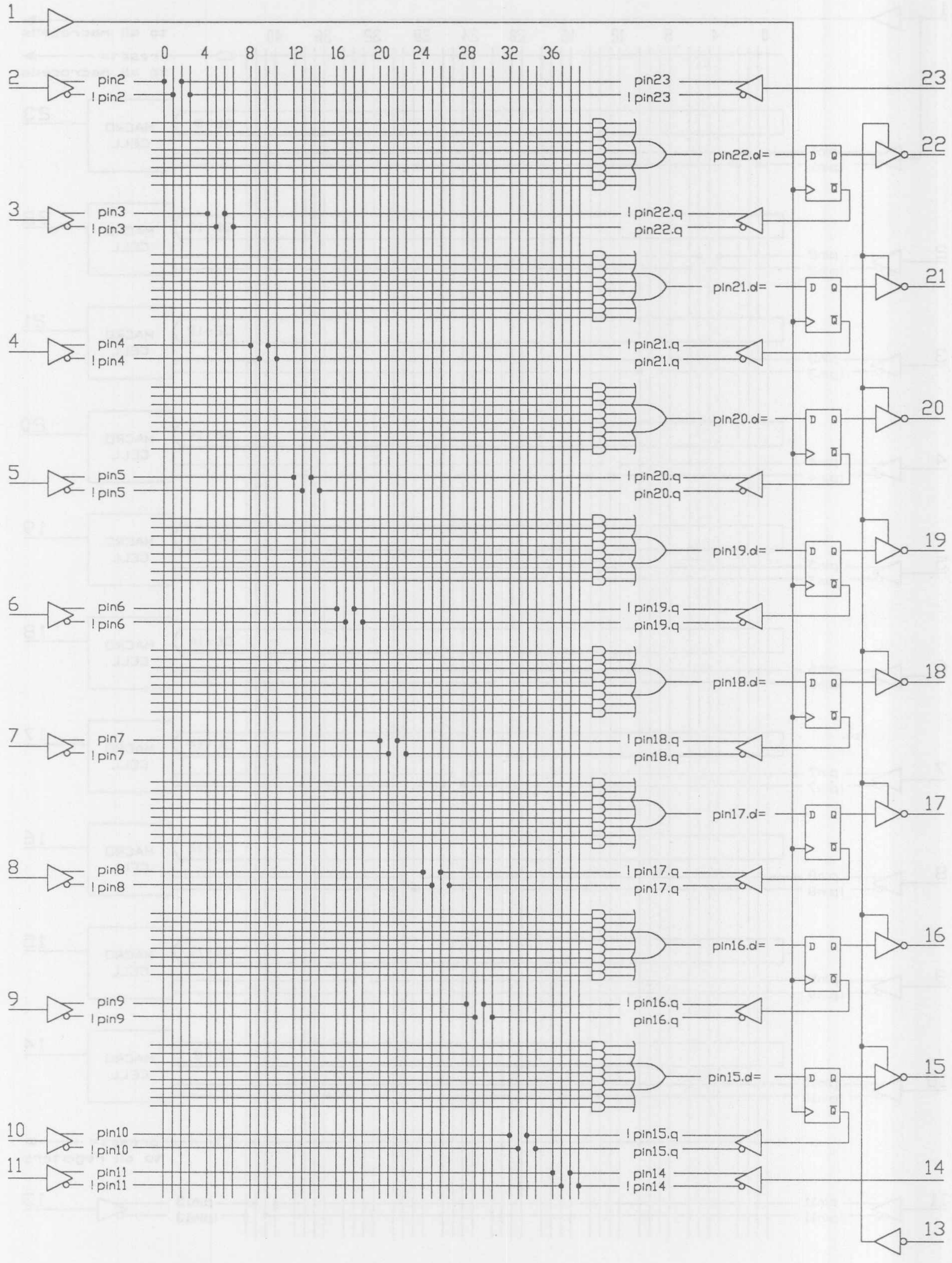
1
2
3
4
5
6
7
8
9
10
11
0 4 8 12 16 20 24 28 32 36
pin2
! pin2
pin3
! pin3
pin4
! pin4
pin5
! pin5
pin6
! pin6
pin7
! pin7
pin8
! pin8
pin9
! pin9
pin10
! pin10
pin11
! pin11
pin23
! pin23
pin22.oe=
! pin22=
pin22
! pin22
pin21.oe=
! pin21=
pin21
! pin21
pin20.d=
! pin20.q
pin20.q
pin19.d=
! pin19.q
pin19.q
pin18.d=
! pin18.q
pin18.q
pin17.d=
! pin17.q
pin17.q
pin16.oe=
! pin16=
pin16
! pin16
pin15.oe=
! pin15=
pin15
! pin15
pin14
! pin14
D
Q
23
22
21
20
19
18
17
16
15
14
13

p20r4

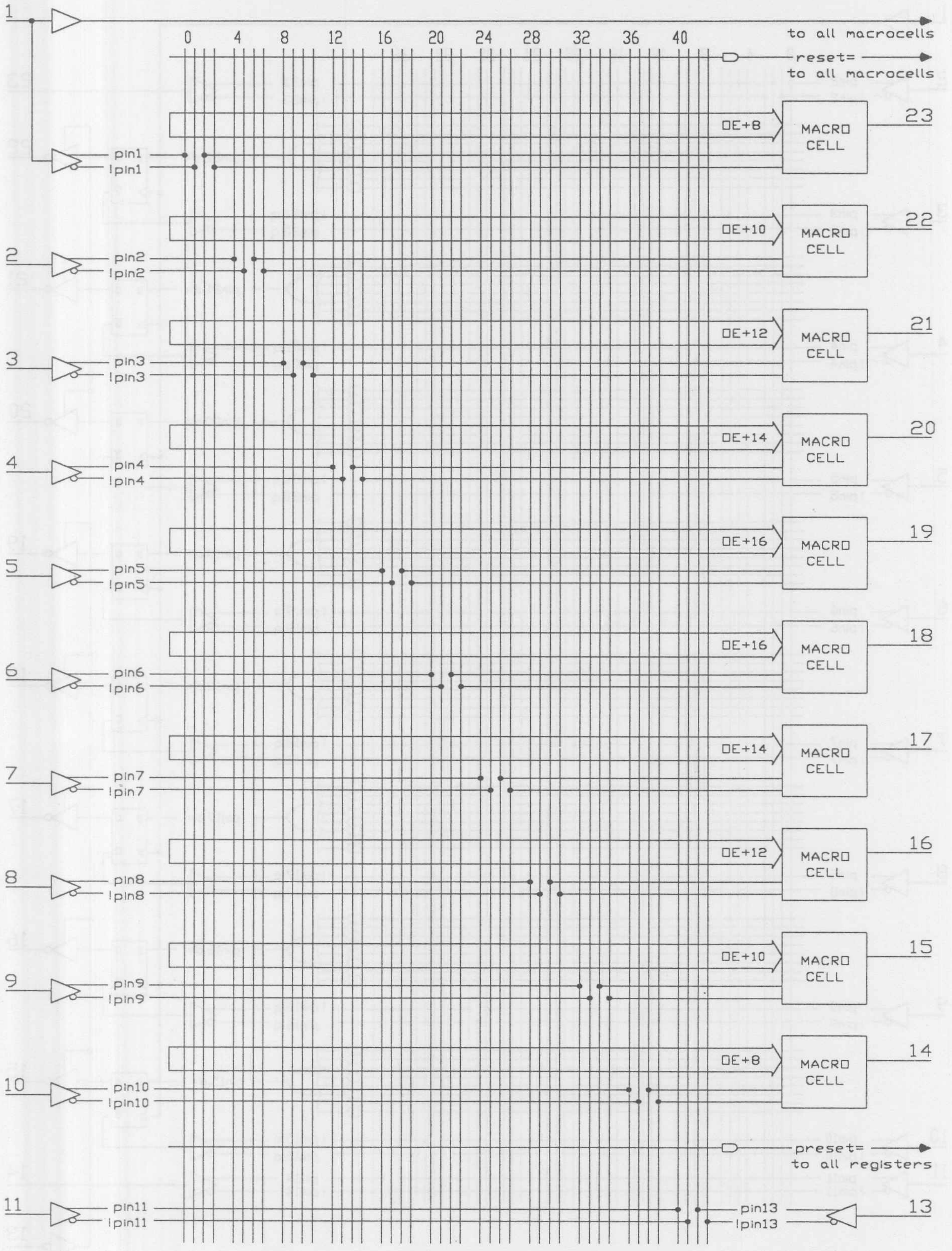


1
2
3
4
5
6
7
8
9
10
11
0
4
8
12
16
20
24
28
32
36
pin2
! pin2
pin3
! pin3
pin4
! pin4
pin5
! pin5
pin6
! pin6
pin7
! pin7
pin8
! pin8
pin9
! pin9
pin10
! pin10
pin11
! pin11
pin23
! pin23
pin22.oe=
! pin22=
pin22
! pin22
pin21.d=
! pin21.q
pin21.q
pin20.d=
! pin20.q
pin20.q
pin19.d=
! pin19.q
pin19.q
pin18.d=
! pin18.q
pin18.q
pin17.d=
! pin17.q
pin17.q
pin16.d=
! pin16.q
pin16.q
pin15.oe=
! pin15=
pin15
! pin15
pin14
! pin14
D
Q
Q̄
23
22
21
20
19
18
17
16
15
14
13

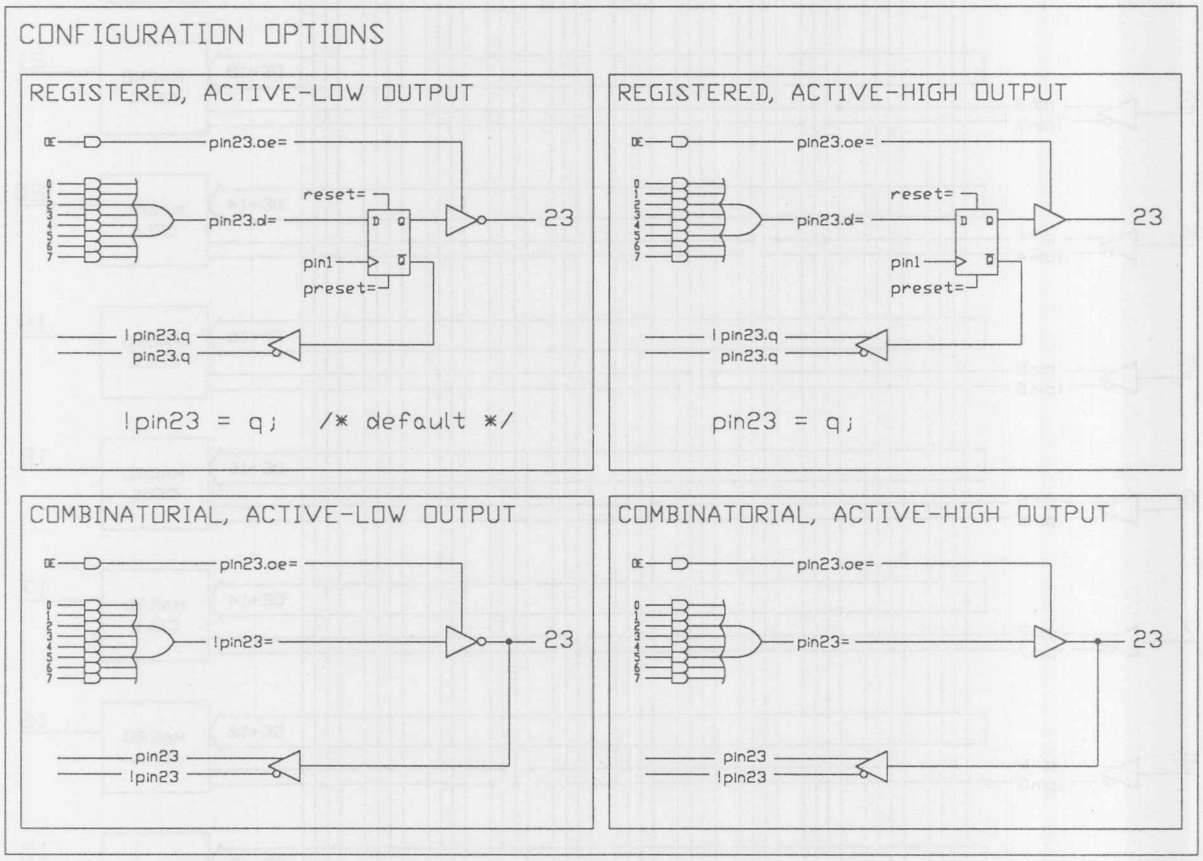
p20r6


1
2
3
4
5
6
7
8
9
10
11
0
4
8
12
16
20
24
28
32
36
pin2
! pin2
pin3
! pin3
pin4
! pin4
pin5
! pin5
pin6
! pin6
pin7
! pin7
pin8
! pin8
pin9
! pin9
pin10
! pin10
pin11
! pin11
pin23
! pin23
pin22.d=
! pin22.q
pin22.q
pin21.d=
! pin21.q
pin21.q
pin20.d=
! pin20.q
pin20.q
pin19.d=
! pin19.q
pin19.q
pin18.d=
! pin18.q
pin18.q
pin17.d=
! pin17.q
pin17.q
pin16.d=
! pin16.q
pin16.q
pin15.d=
! pin15.q
pin15.q
pin14
! pin14
D
Q
23
22
21
20
19
18
17
16
15
14
13

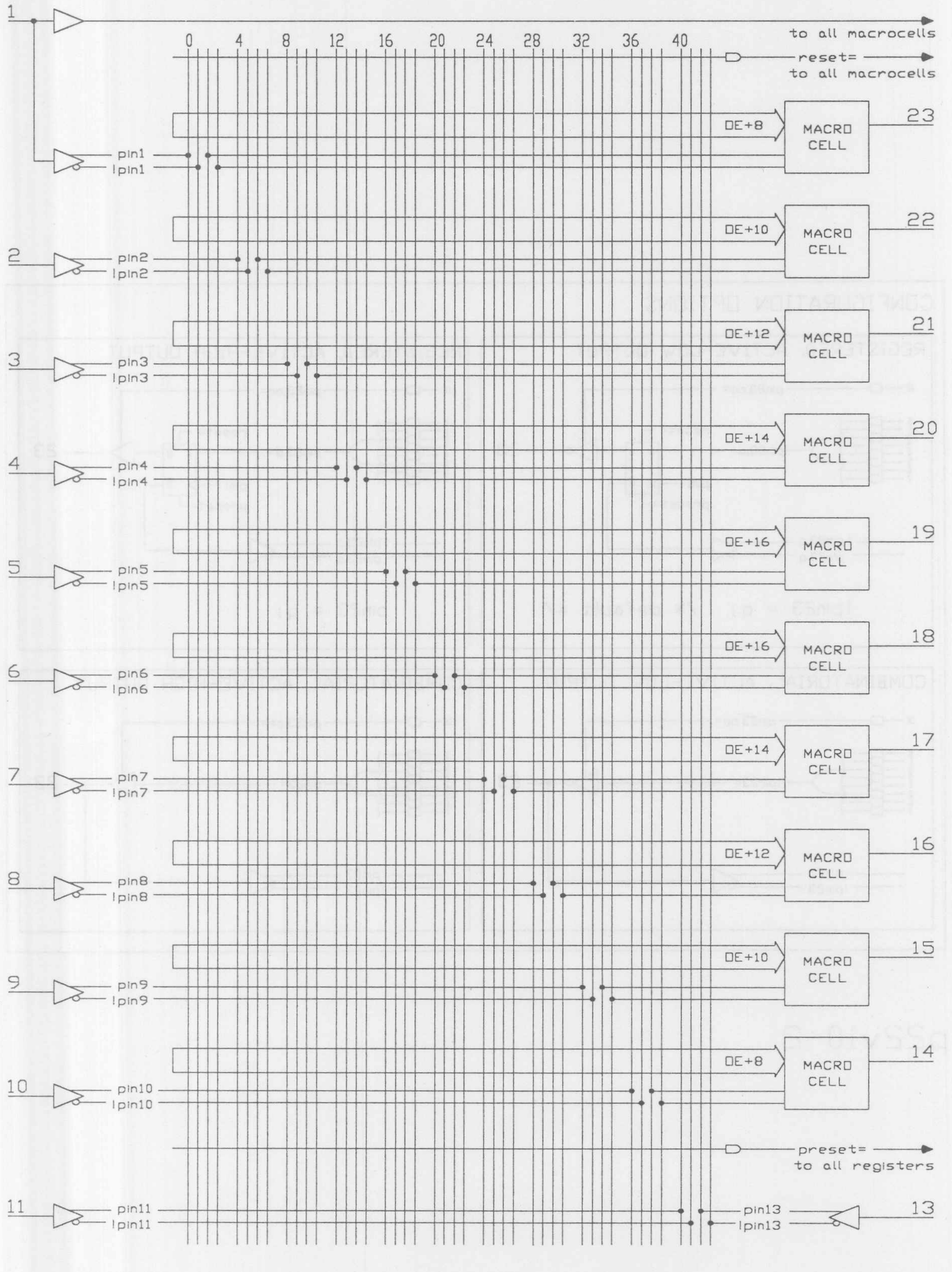
p20r8


1
to all macrocells
0
4
8
12
16
20
24
28
32
36
40
reset=
to all macrocells
OE+8
MACRO CELL
23
pin1
!pin1
OE+10
MACRO CELL
22
2
pin2
!pin2
OE+12
MACRO CELL
21
3
pin3
!pin3
OE+14
MACRO CELL
20
4
pin4
!pin4
OE+16
MACRO CELL
19
5
pin5
!pin5
OE+16
MACRO CELL
18
6
pin6
!pin6
OE+14
MACRO CELL
17
7
pin7
!pin7
OE+12
MACRO CELL
16
8
pin8
!pin8
OE+10
MACRO CELL
15
9
pin9
!pin9
OE+8
MACRO CELL
14
10
pin10
!pin10
preset=
to all registers
11
pin11
!pin11
pin13
!pin13
13

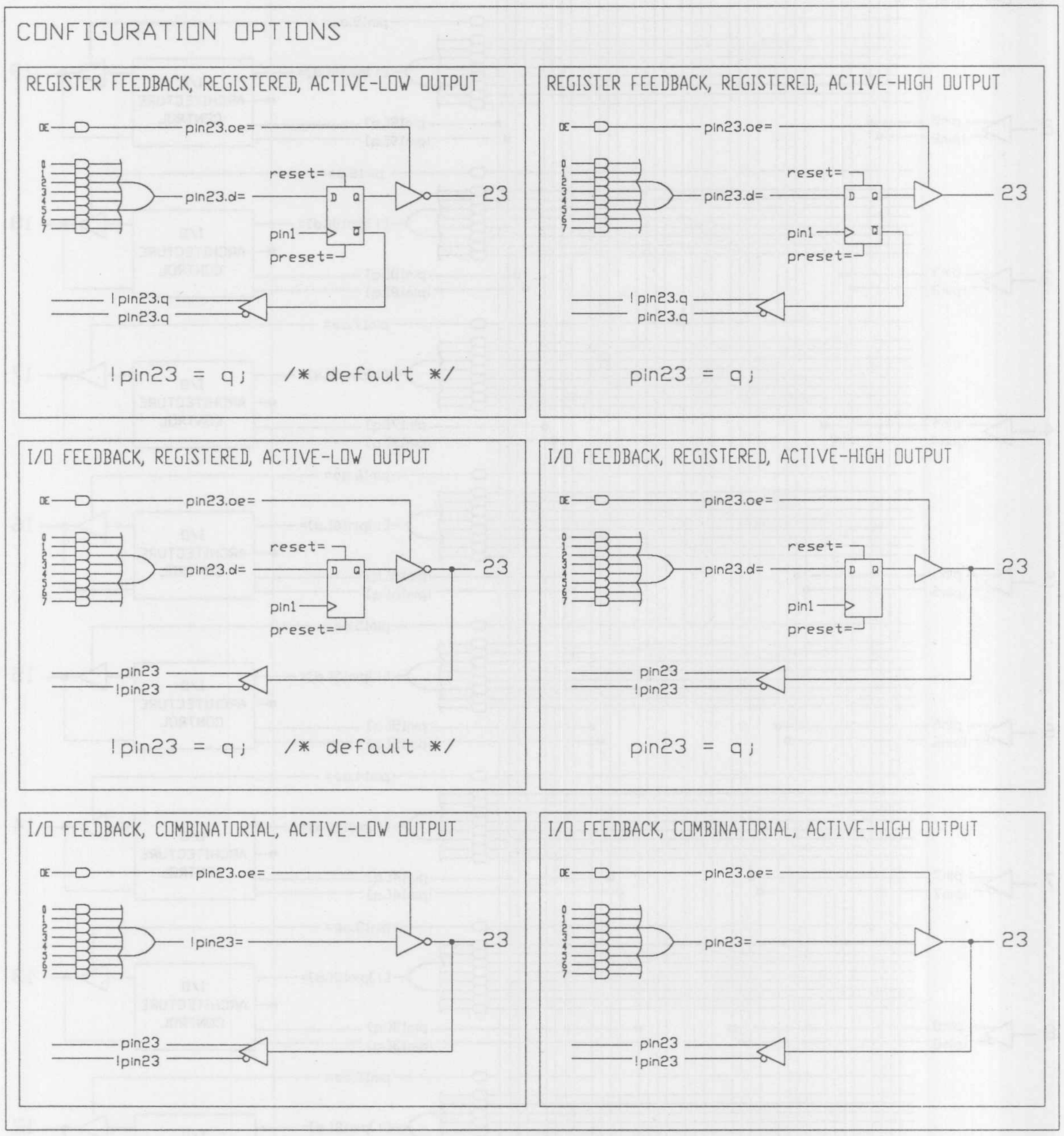
p22v10-1


CONFIGURATION OPTIONS
REGISTERED, ACTIVE-LOW OUTPUT
OE
pin23.oe=
0
1
2
3
4
5
6
7
pin23.d=
reset=
D
Q
pin1
Q̄
preset=
23
!pin23.q
pin23.q
!pin23 = q; /* default */
REGISTERED, ACTIVE-HIGH OUTPUT
OE
pin23.oe=
0
1
2
3
4
5
6
7
pin23.d=
reset=
D
Q
pin1
Q̄
preset=
23
!pin23.q
pin23.q
pin23 = q;
COMBINATORIAL, ACTIVE-LOW OUTPUT
OE
pin23.oe=
0
1
2
3
4
5
6
7
!pin23=
23
pin23
!pin23
COMBINATORIAL, ACTIVE-HIGH OUTPUT
OE
pin23.oe=
0
1
2
3
4
5
6
7
pin23=
23
pin23
!pin23

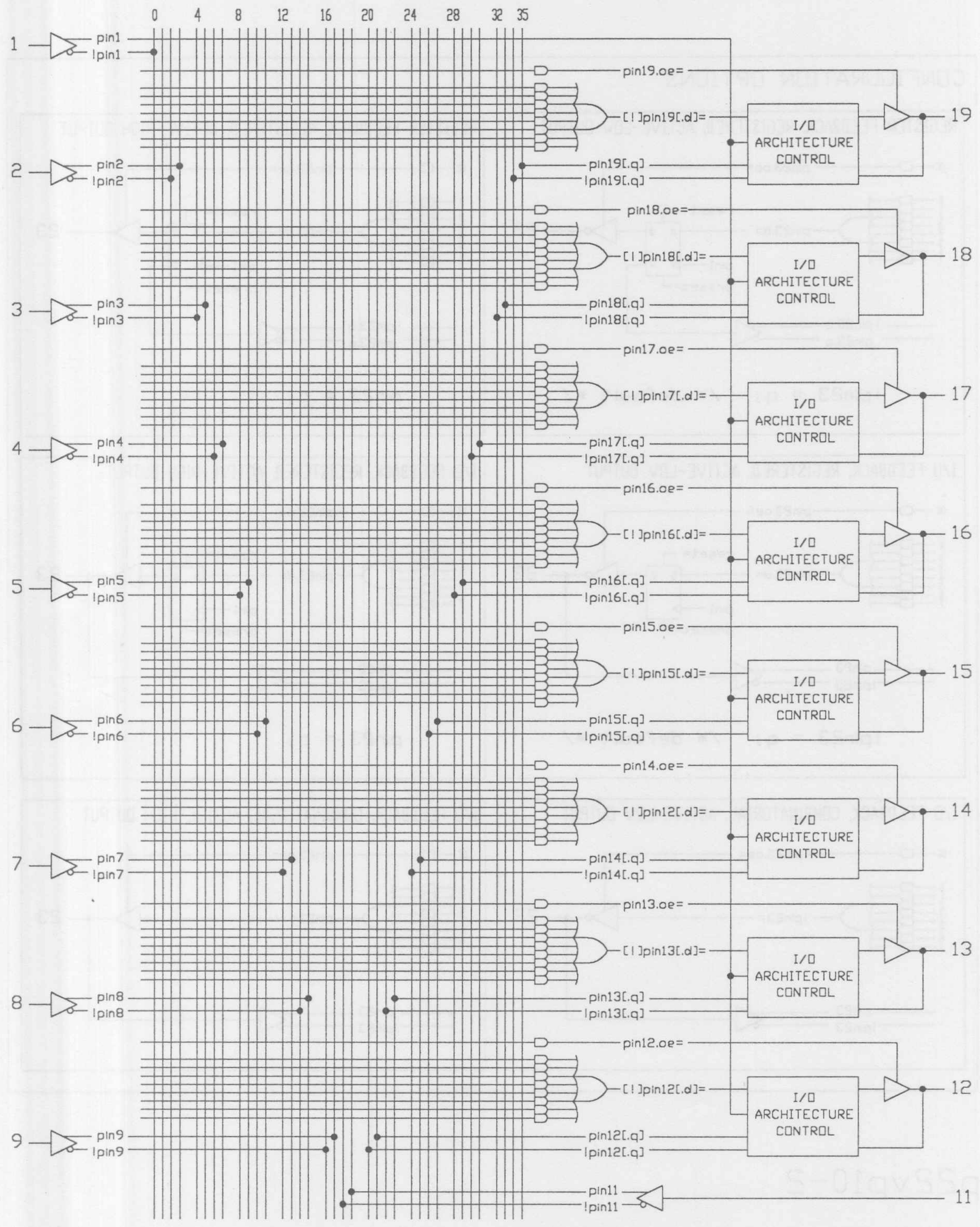
p22v10-2


1
0
4
8
12
16
20
24
28
32
36
40
to all macrocells
reset=
to all macrocells
OE+8
MACRO CELL
23
pin1
!pin1
OE+10
22
2
pin2
!pin2
OE+12
21
3
pin3
!pin3
OE+14
20
4
pin4
!pin4
OE+16
19
5
pin5
!pin5
OE+16
18
6
pin6
!pin6
OE+14
17
7
pin7
!pin7
OE+12
16
8
pin8
!pin8
OE+10
15
9
pin9
!pin9
OE+8
14
10
pin10
!pin10
preset=
to all registers
11
pin11
!pin11
pin13
!pin13
13

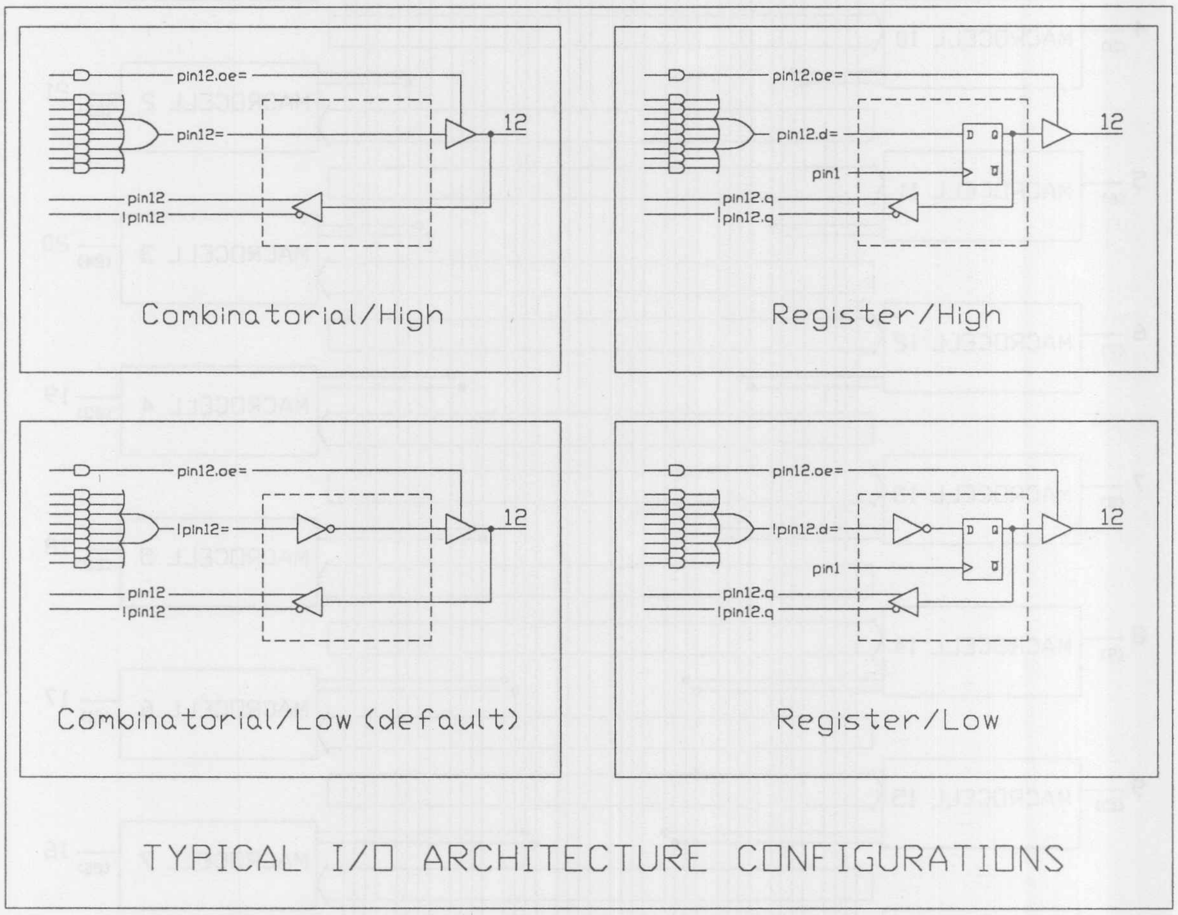
p22vp10-1


CONFIGURATION OPTIONS
REGISTER FEEDBACK, REGISTERED, ACTIVE-LOW OUTPUT
OE
pin23.oe=
pin23.d=
reset=
D Q
pin1
preset=
23
!pin23.q
pin23.q
!pin23 = q; /* default */
REGISTER FEEDBACK, REGISTERED, ACTIVE-HIGH OUTPUT
pin23 = q;
I/O FEEDBACK, REGISTERED, ACTIVE-LOW OUTPUT
pin23
!pin23
!pin23 = q; /* default */
I/O FEEDBACK, REGISTERED, ACTIVE-HIGH OUTPUT
pin23 = q;
I/O FEEDBACK, COMBINATORIAL, ACTIVE-LOW OUTPUT
!pin23=
I/O FEEDBACK, COMBINATORIAL, ACTIVE-HIGH OUTPUT
pin23=

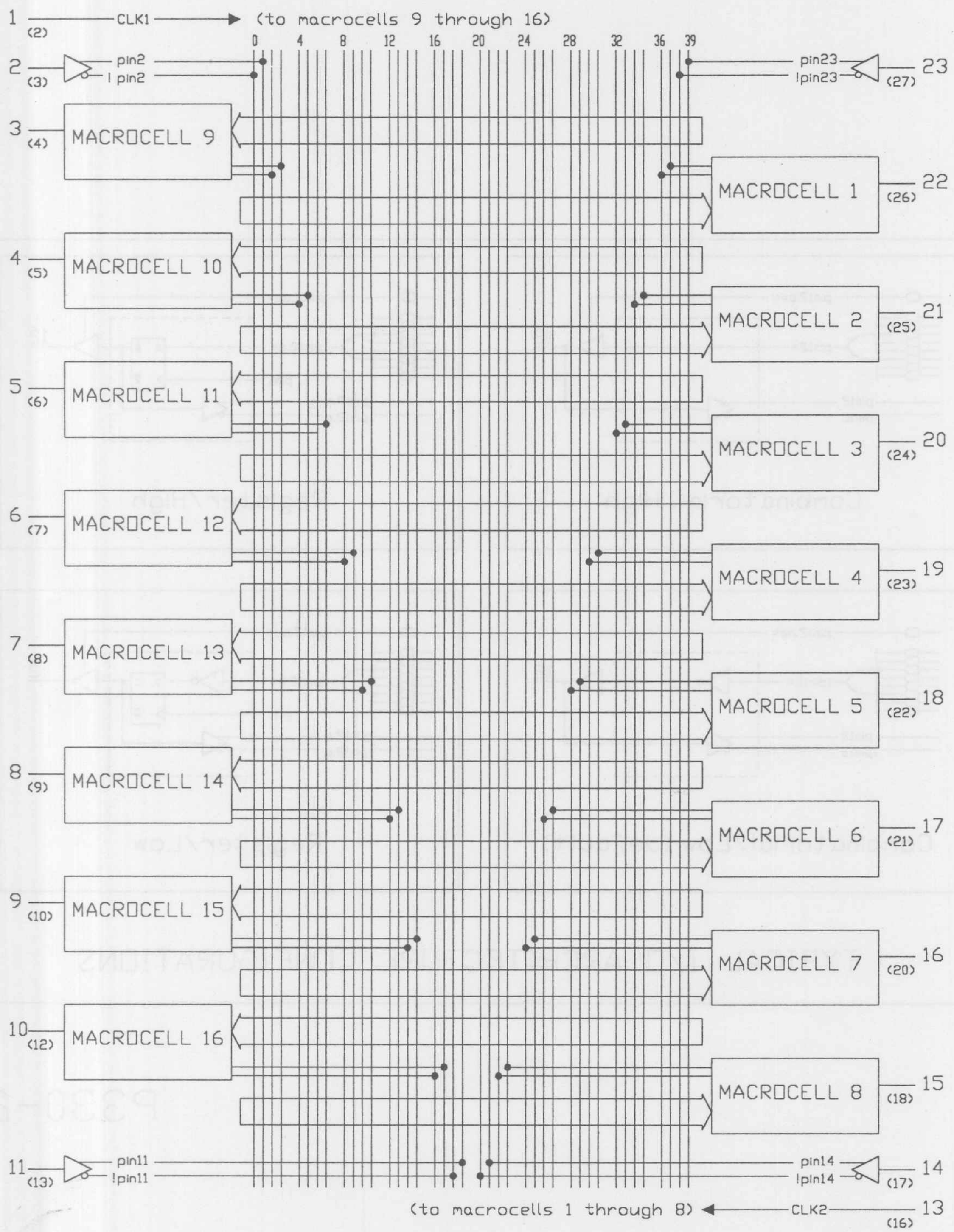
p22vp10-2


0
4
8
12
16
20
24
28
32
35
1
pin1
!pin1
2
pin2
!pin2
3
pin3
!pin3
4
pin4
!pin4
5
pin5
!pin5
6
pin6
!pin6
7
pin7
!pin7
8
pin8
!pin8
9
pin9
!pin9
pin11
!pin11
11
pin19.oe=
[!]pin19[.d]=
pin19[.q]
!pin19[.q]
19
pin18.oe=
[!]pin18[.d]=
pin18[.q]
!pin18[.q]
18
pin17.oe=
[!]pin17[.d]=
pin17[.q]
!pin17[.q]
17
pin16.oe=
[!]pin16[.d]=
pin16[.q]
!pin16[.q]
16
pin15.oe=
[!]pin15[.d]=
pin15[.q]
!pin15[.q]
15
pin14.oe=
[!]pin12[.d]=
pin14[.q]
!pin14[.q]
14
pin13.oe=
[!]pin13[.d]=
pin13[.q]
!pin13[.q]
13
pin12.oe=
[!]pin12[.d]=
pin12[.q]
!pin12[.q]
12
I/O ARCHITECTURE CONTROL

P330-1

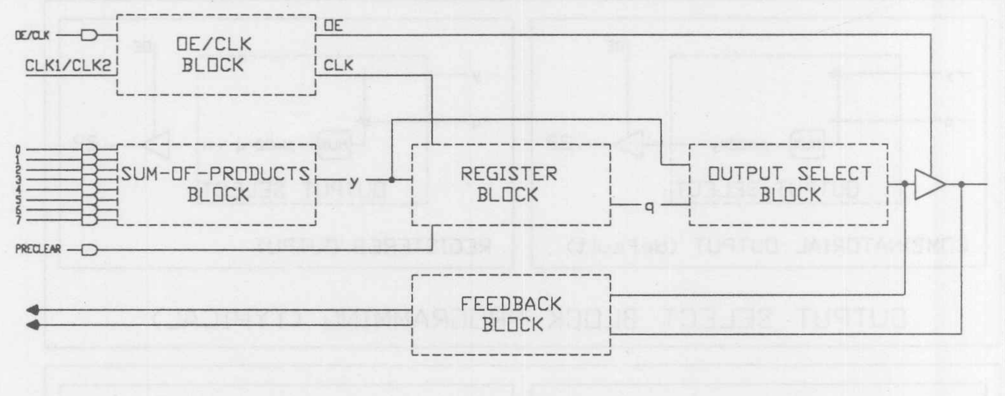

pin12.oe=
pin12=
12
pin12
!pin12
Combinatorial/High
pin12.oe=
pin12.d=
pin1
pin12.q
!pin12.q
12
Register/High
pin12.oe=
!pin12=
12
pin12
!pin12
Combinatorial/Low (default)
pin12.oe=
!pin12.d=
pin1
pin12.q
!pin12.q
12
Register/Low
TYPICAL I/O ARCHITECTURE CONFIGURATIONS

P330-2

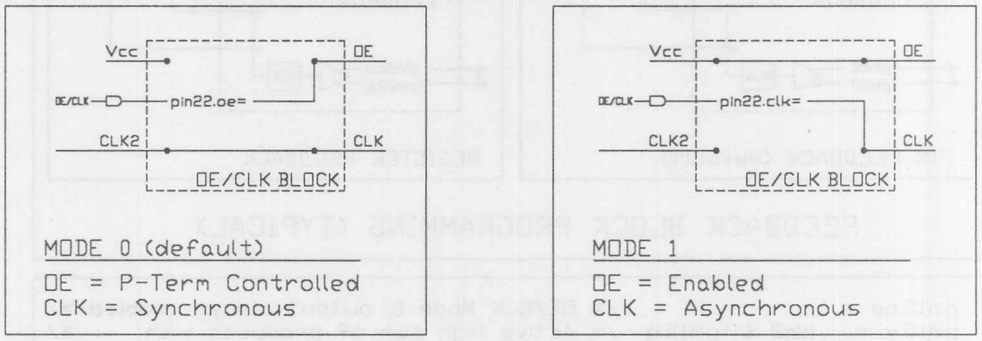


1
(2)
CLK1
(to macrocells 9 through 16)
0
4
8
12
16
20
24
28
32
36
39
2
(3)
pin2
! pin2
pin23
!pin23
23
(27)
3
(4)
MACROCELL 9
MACROCELL 1
22
(26)
4
(5)
MACROCELL 10
MACROCELL 2
21
(25)
5
(6)
MACROCELL 11
MACROCELL 3
20
(24)
6
(7)
MACROCELL 12
MACROCELL 4
19
(23)
7
(8)
MACROCELL 13
MACROCELL 5
18
(22)
8
(9)
MACROCELL 14
MACROCELL 6
17
(21)
9
(10)
MACROCELL 15
MACROCELL 7
16
(20)
10
(12)
MACROCELL 16
MACROCELL 8
15
(18)
11
(13)
pin11
!pin11
pin14
!pin14
14
(17)
(to macrocells 1 through 8)
CLK2
13
(16)

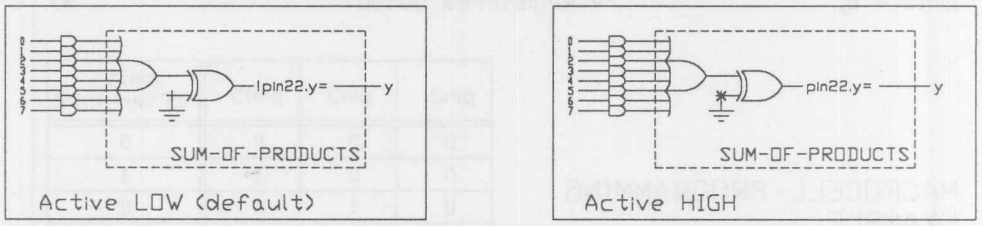
P630-1



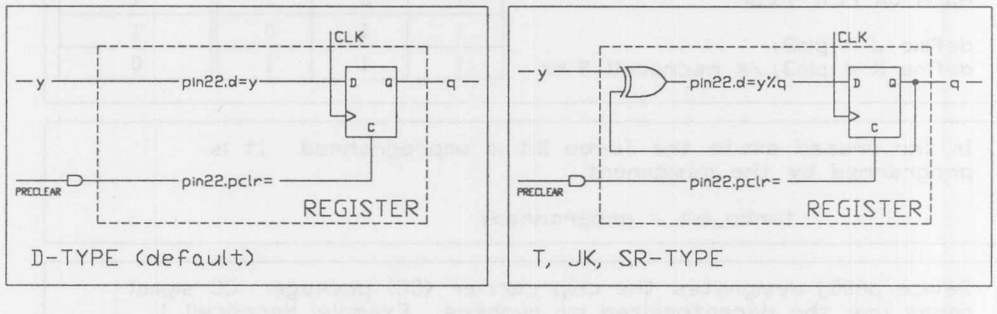
MACROCELL FUNCTIONAL ARCHITECTURE



OE/CLK BLOCK PROGRAMMING (TYPICAL)

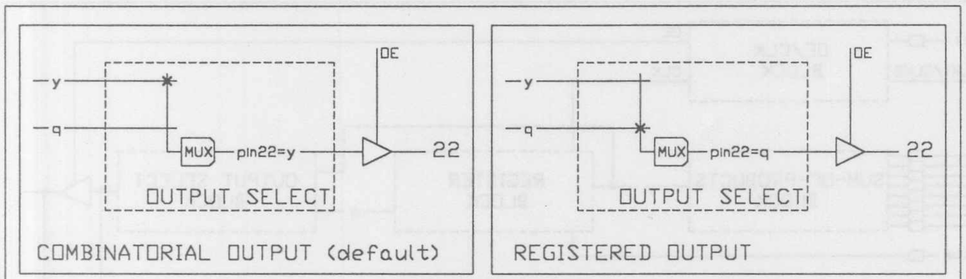


SUM-OF-PRODUCTS BLOCK PROGRAMMING (TYPICAL)

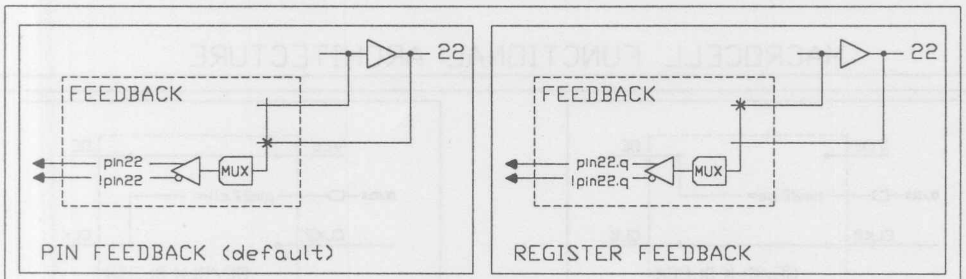


REGISTER BLOCK PROGRAMMING (TYPICAL)

P630-2



OUTPUT SELECT BLOCK PROGRAMMING (TYPICAL)



FEEDBACK BLOCK PROGRAMMING (TYPICAL)

```
pin15.oe = 1;              /* OE/CLK Mode 0, output always enabled */
pin15.y =     pin2 & !pin15.q   /* Active High sum of products with      */
            | pin3 & pin15.q;   /* register feedback selection for pin15 */
                           /* and pin feedback selection for pin3    */
signal pin15.d = y % q ;   /* T, JK, SR-TYPE Register              */
pin15 = q;                 /* Registered Output                    */
```

MACROCELL PROGRAMMING EXAMPLE:

MACROCELL 8 CONFIGURED
AS A JK FLIP-FLOP

```
define J = pin2;
define K = pin3; /* macrocell 9 */
```

pin2	pin3	pin15	pin15 after clk2
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

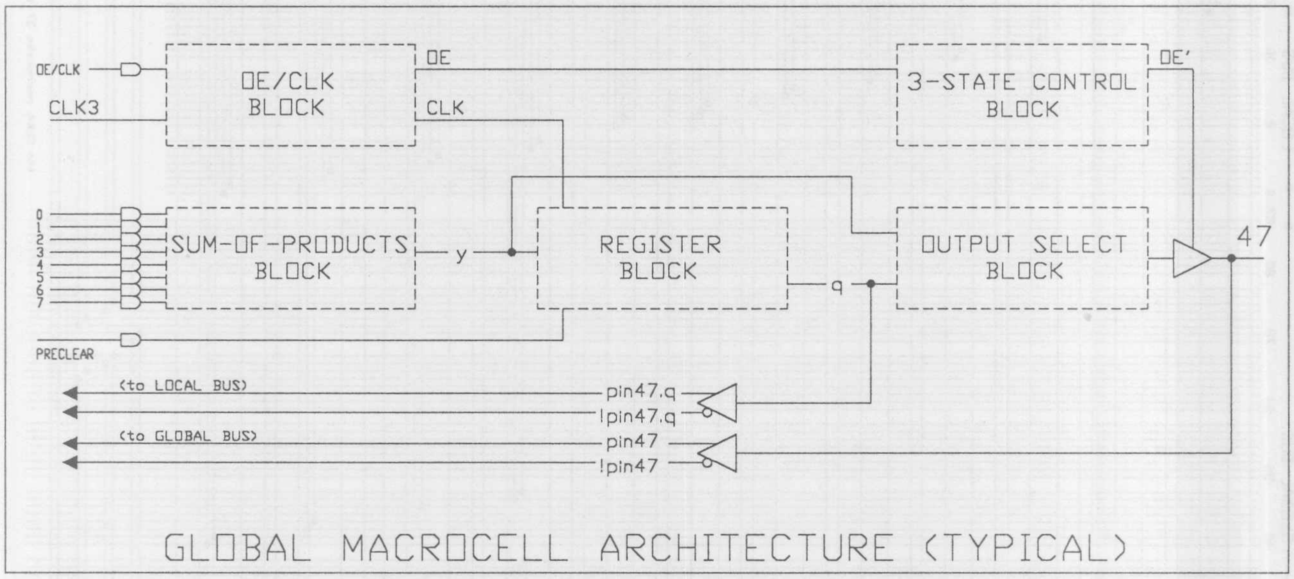
In the erased state the Turbo Bit is unprogrammed. It is programmed by the assignment

```
turbo_bit = programmed;
```

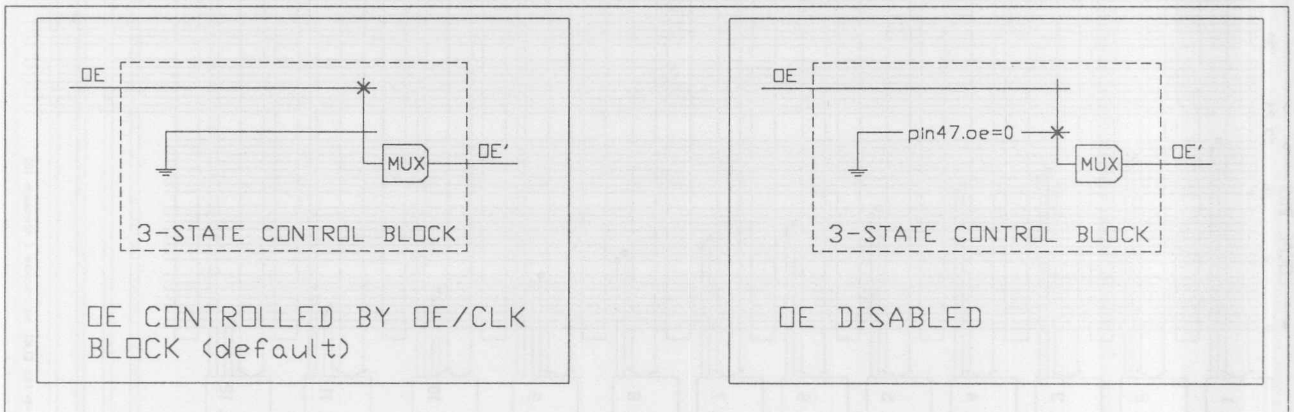
Device p600j designates the chip carrier (CC) package. CC signal names use the parenthesized pin numbers. Example: Macrocell 1 output select is pin26 = y for the CC package.

p630-3

The proLogic Simulator uses pin 1 as GND and pin 52 as Vcc.



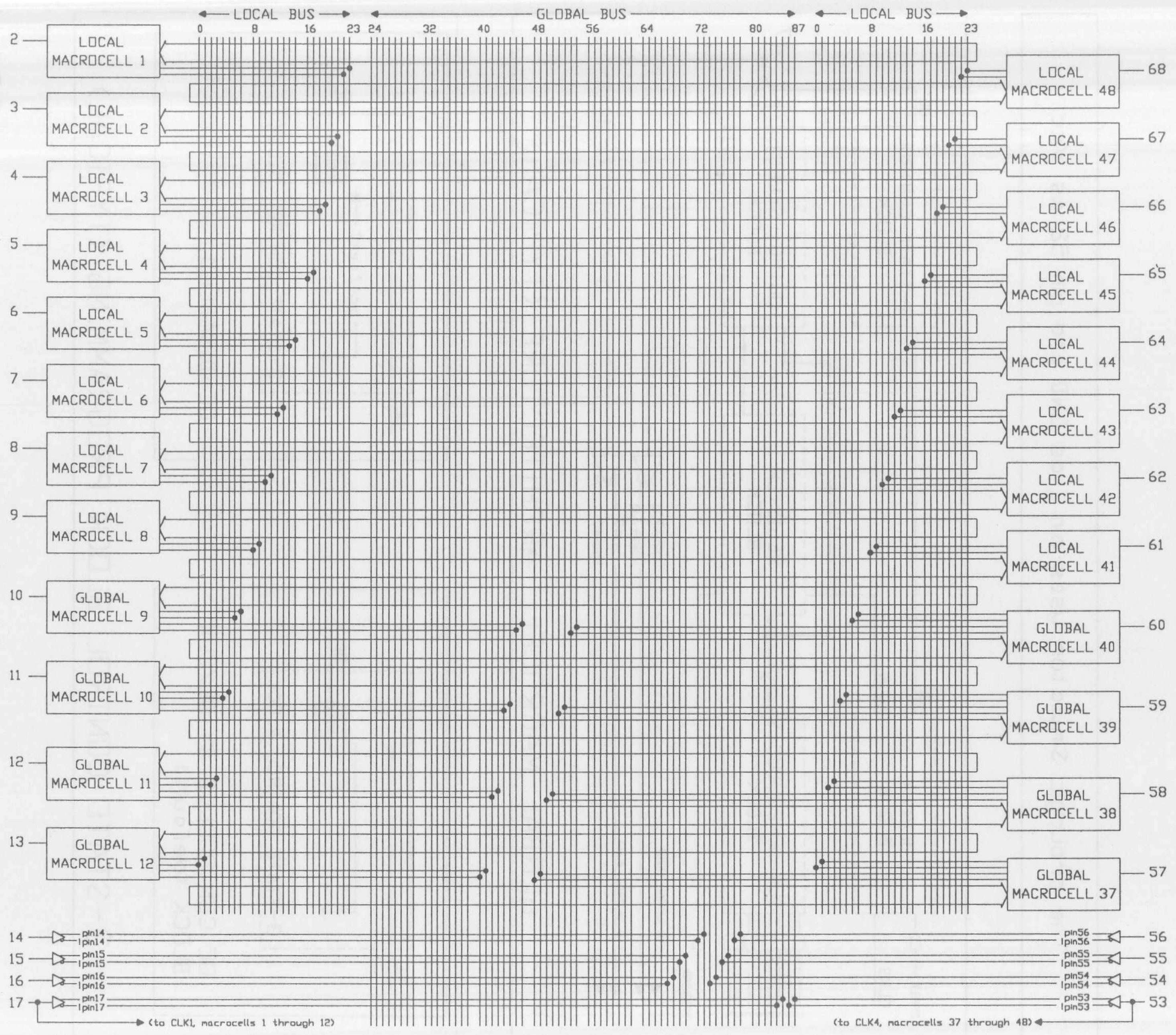
GLOBAL MACROCELL ARCHITECTURE (TYPICAL)



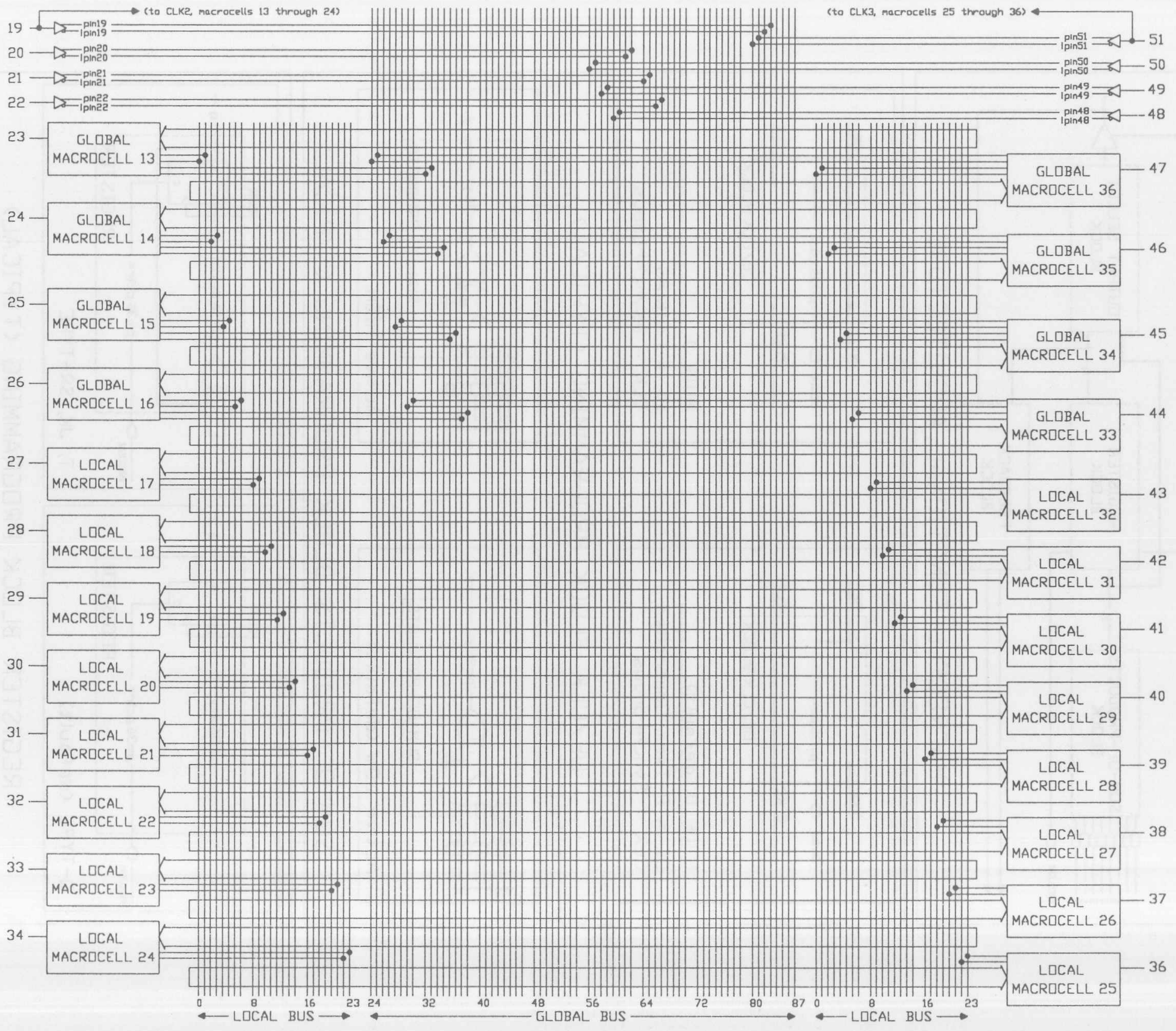
3-STATE CONTROL BLOCK PROGRAMMING (TYPICAL)

p1830-1

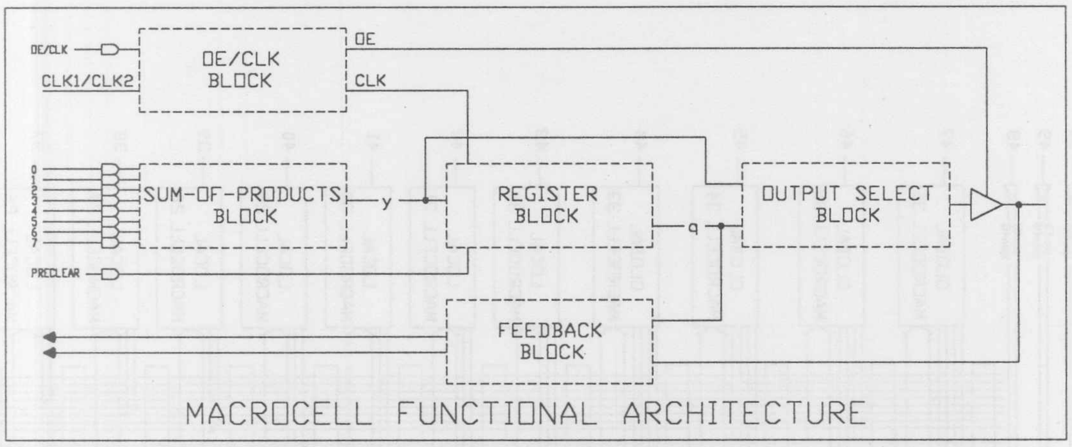
p1830-2

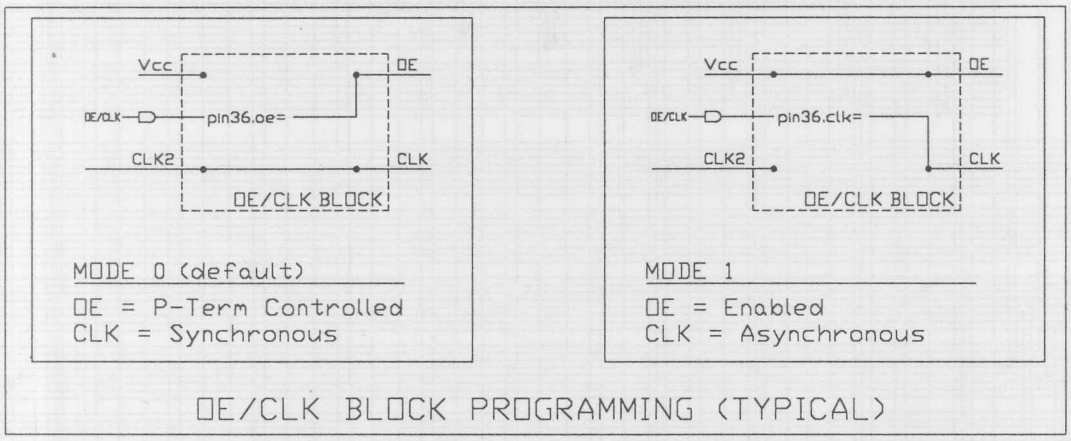


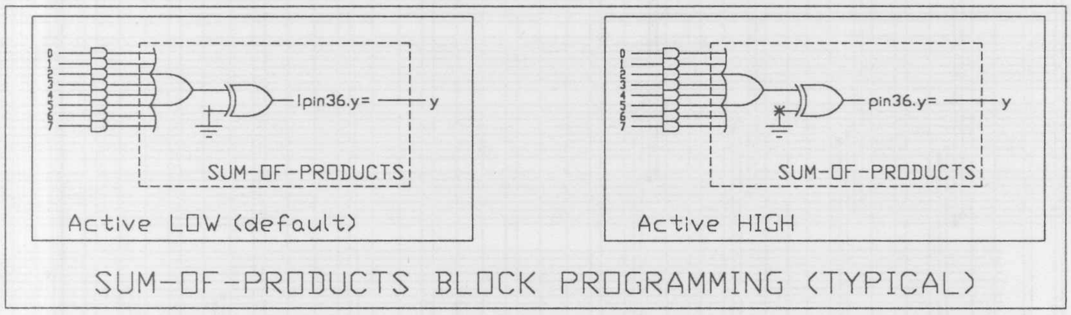
LOCAL BUS
GLOBAL BUS
LOCAL BUS
0 8 16 23 24 32 40 48 56 64 72 80 87 0 8 16 23
LOCAL MACROCELL 1
LOCAL MACROCELL 2
LOCAL MACROCELL 3
LOCAL MACROCELL 4
LOCAL MACROCELL 5
LOCAL MACROCELL 6
LOCAL MACROCELL 7
LOCAL MACROCELL 8
GLOBAL MACROCELL 9
GLOBAL MACROCELL 10
GLOBAL MACROCELL 11
GLOBAL MACROCELL 12
LOCAL MACROCELL 48
LOCAL MACROCELL 47
LOCAL MACROCELL 46
LOCAL MACROCELL 45
LOCAL MACROCELL 44
LOCAL MACROCELL 43
LOCAL MACROCELL 42
LOCAL MACROCELL 41
GLOBAL MACROCELL 40
GLOBAL MACROCELL 39
GLOBAL MACROCELL 38
GLOBAL MACROCELL 37
2 3 4 5 6 7 8 9 10 11 12 13
68 67 66 65 64 63 62 61 60 59 58 57
14 pin14 !pin14
15 pin15 !pin15
16 pin16 !pin16
17 pin17 !pin17
56 pin56 !pin36
55 pin55 !pin55
54 pin54 !pin54
53 pin53 !pin53
(to CLK1, macrocells 1 through 12)
(to CLK4, macrocells 37 through 48)

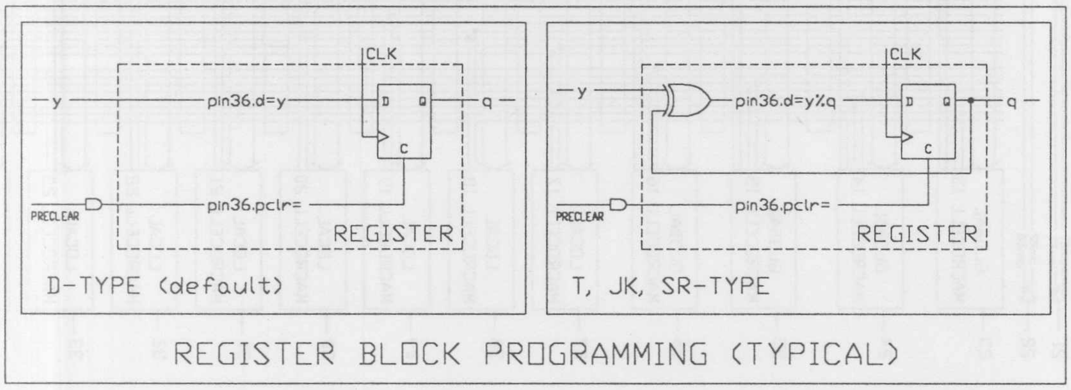

(to CLK2, macrocells 13 through 24)
(to CLK3, macrocells 25 through 36)
19
20
21
22
pin19
Ipin19
pin20
Ipin20
pin21
Ipin21
pin22
Ipin22
pin51
Ipin51
pin50
Ipin50
pin49
Ipin49
pin48
Ipin48
51
50
49
48
23 GLOBAL MACROCELL 13
24 GLOBAL MACROCELL 14
25 GLOBAL MACROCELL 15
26 GLOBAL MACROCELL 16
27 LOCAL MACROCELL 17
28 LOCAL MACROCELL 18
29 LOCAL MACROCELL 19
30 LOCAL MACROCELL 20
31 LOCAL MACROCELL 21
32 LOCAL MACROCELL 22
33 LOCAL MACROCELL 23
34 LOCAL MACROCELL 24
GLOBAL MACROCELL 36 47
GLOBAL MACROCELL 35 46
GLOBAL MACROCELL 34 45
GLOBAL MACROCELL 33 44
LOCAL MACROCELL 32 43
LOCAL MACROCELL 31 42
LOCAL MACROCELL 30 41
LOCAL MACROCELL 29 40
LOCAL MACROCELL 28 39
LOCAL MACROCELL 27 38
LOCAL MACROCELL 26 37
LOCAL MACROCELL 25 36
0 8 16 23
LOCAL BUS
24 32 40 48 56 64 72 80 87
GLOBAL BUS
0 8 16 23
LOCAL BUS

A-DEp1830-3

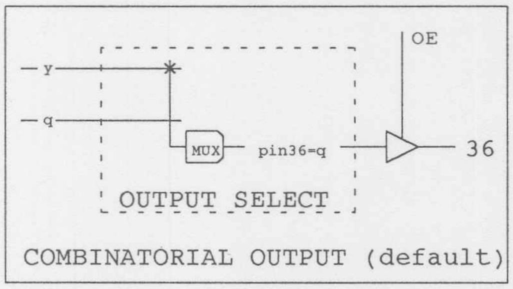

OE/CLK
CLK1/CLK2
OE/CLK BLOCK
OE
CLK
0 1 2 3 4 5 6 7
SUM-OF-PRODUCTS BLOCK
y
REGISTER BLOCK
q
OUTPUT SELECT BLOCK
PRECLEAR
FEEDBACK BLOCK
MACROCELL FUNCTIONAL ARCHITECTURE

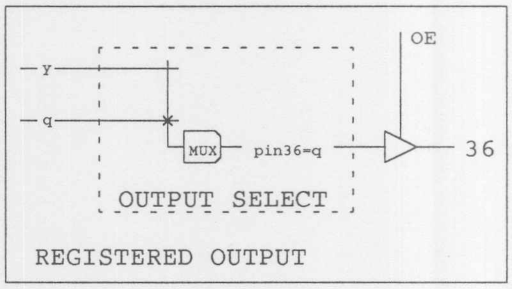

Vcc
OE
OE/CLK
pin36.oe=
CLK2
CLK
OE/CLK BLOCK
MODE 0 (default)
OE = P-Term Controlled
CLK = Synchronous
Vcc
OE
OE/CLK
pin36.clk=
CLK2
CLK
OE/CLK BLOCK
MODE 1
OE = Enabled
CLK = Asynchronous
OE/CLK BLOCK PROGRAMMING (TYPICAL)


0 1 2 3 4 5 6 7
!pin36.y=
y
SUM-OF-PRODUCTS
Active LOW (default)
0 1 2 3 4 5 6 7
pin36.y=
y
SUM-OF-PRODUCTS
Active HIGH
SUM-OF-PRODUCTS BLOCK PROGRAMMING (TYPICAL)

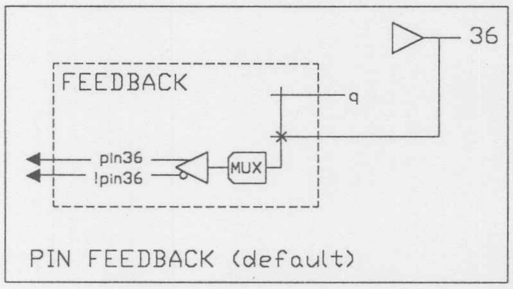

CLK
y
pin36.d=y
D
Q
C
q
PRECLEAR
pin36.pclr=
REGISTER
D-TYPE (default)
CLK
y
pin36.d=y%q
D
Q
C
q
PRECLEAR
pin36.pclr=
REGISTER
T, JK, SR-TYPE
REGISTER BLOCK PROGRAMMING (TYPICAL)

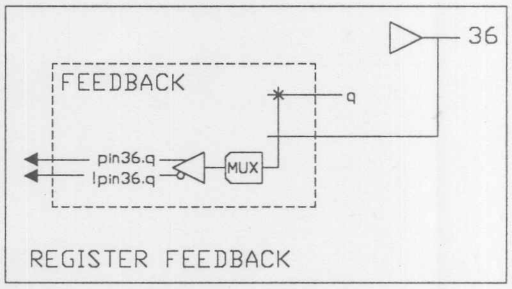
p1830-4





OUTPUT SELECT BLOCK PROGRAMMING (TYPICAL)





FEEDBACK BLOCK PROGRAMMING (TYPICAL)

```
pin36.oe = 1;                          /* OE/CLK Mode 0, output always enable    */
pin36.y = pin47 & !pin36.q             /* Active High sum of products with       */
        | pin48 &  pin36.q;            /* register feedback selection for pin36  */
                                       /* and pin feedback selection for pin47   */
signal pin36.d = y % q;                /* T, JK, SR-TYPE register                */
pin36 = q;                             /* Registered Output                      */
```

MACROCELL PROGRAMMING EXAMPLE

MACROCELL 25 CONFIGURED AS A JK FLIP-FLOP

```
define J = pin47; /* macrocell 36 */
define K = pin48;
```

pin47	pin48	pin36	pin36 after clk2
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

In the erased state the Turbo Bit is unprogrammed. It is programmed by the assignment

```
        turbo_bit = programmed;
```

p1830-5